# XAMARIN.FORMS SUCCINCTLY

*BY* **ALESSANDRO DEL SOLE**

SUCCINCTLY EBOOK SERIES

Syncfusion®

# Xamarin.Forms Succinctly

**By**

**Alessandro Del Sole**

Foreword by Daniel Jebaraj

ISBN: 978-1-64200-212-6

**Technical Reviewer:** James McCaffrey
**Copy Editor:** Courtney Wright
**Acquisitions Coordinator:** Tres Watkins, VP of content, Syncfusion, Inc.
**Proofreader:** Jacqueline Bieringer, content producer, Syncfusion, Inc.

# Table of Contents

# The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

## Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

## Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

## The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

## The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

## Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

## Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

## Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

# Note on the Fourth Edition

This is the fourth edition of *Xamarin.Forms Succinctly* by Alessandro Del Sole. This update covers features added from Xamarin.Forms version 3.5 to version 5.0, the XAML code editor, tools for developing iOS applications, and using native Android and iOS controls without writing custom APIs.

# About the Author

Alessandro Del Sole is a Xamarin Certified Mobile Developer and has been a Microsoft MVP since 2008. Awarded MVP of the Year in 2009, 2010, 2011, 2012, and 2014, he is internationally considered a Visual Studio expert and a .NET authority. Alessandro has authored many printed books and ebooks on programming with Visual Studio, including *Visual Studio 2017 Succinctly*, *Visual Basic 2015 Unleashed*, and *Visual Studio Code Succinctly*. He has written tons of technical articles about .NET, Visual Studio, and other Microsoft technologies in Italian and English for many developer portals, including MSDN Magazine and the Visual Basic Developer Center from Microsoft. He is a frequent speaker at Italian conferences, and he has released a number of Windows Store apps. He has also produced a number of instructional videos in both English and Italian. Alessandro works as a senior software engineer for Fresenius Medical Care, focusing on building mobile apps with Xamarin in the healthcare market. You can follow him on Twitter at @progalex.

# Introduction

For mobile app developers and companies that want to be represented on the market by mobile applications, the need to publish Android, iOS, and Windows versions of apps has dramatically increased in the last few years. For companies that have always worked with native platforms and development tools, this might not be a problem. It is a problem, though, for companies that have always built software with .NET, C#, and, more generally, with the Microsoft stack. A company might therefore hire specialized developers or wait for existing developers to attain the necessary skills and knowledge to work with native platforms, but in both cases, there are huge costs and risks with timing. The ideal solution is that developers could reuse their existing .NET and C# skills to build native mobile apps. This is where Xamarin comes in.

In this book, you will learn how Xamarin.Forms allows for cross-platform development, letting you create mobile and desktop apps for Android, iOS, Tizen devices, macOS, and Windows from a single C# codebase, and therefore reuse your existing .NET skills. You will learn how Xamarin.Forms solutions are made, what makes it possible to share code, how to create the user interface, how to organize controls within containers, and how to implement navigation between pages. You will also leverage advanced techniques, such as data binding and accessing native APIs from cross-platform code.

It is worth mentioning that Xamarin.Forms also supports the F# programming language, but C# is clearly the most common choice, and therefore, all the explanations and examples will be provided based on C#. In Visual Studio 2019, Xamarin.Forms only supports UWP for Windows development. For this reason, when I refer to Windows from now on, I mean Windows 10 and the Universal Windows Platform, not the previous versions.

I will assume you have at least a basic knowledge of C# and the Visual Studio IDE. I also suggest you bookmark the official Xamarin documentation portal for quick reference. The source code for this book is available on GitHub. File names are self-explanatory so that it's easier for you to follow the examples, stored under folders whose names match the chapter number. Before you start writing code, you need to set up your development environment. This is the topic of the first chapter.

# Chapter 1 Getting Started with Xamarin.Forms

Before you start writing mobile apps with Xamarin.Forms, you first need to understand the state of mobile app development today and how Xamarin fits into it. Also, you need to set up your development environment to be able to build, test, debug, and deploy your apps to Android, iOS, and Windows devices. This chapter introduces Xamarin as a set of tools and services, introduces Xamarin.Forms as the platform you will use, and then presents the tools and hardware you need for real-world development.

## Introducing Xamarin and Xamarin.Forms

Xamarin is the name of a company that Microsoft acquired in 2016 and, at the same time, the name of a set of development tools and services that developers can use to build native apps for iOS, Android, and Windows in C#. Xamarin's main goal is to make it easier for .NET developers to build native apps for Android, iOS, and Windows reusing their existing skills. The reason behind this goal is simple: building apps for Android requires you to know Java and Android Studio or Eclipse; building apps for iOS requires you to know Objective-C or Swift and Xcode; and building apps for Windows requires you to know C# and Visual Studio. As an existing .NET developer—whether you are experienced or a beginner—getting to know all the possible platforms, languages, and development environments is extremely difficult, and costs are extremely high.

Xamarin allows you to build native apps with C#, based on a cross-platform, open-source porting of the .NET Framework called Mono. From a development point of view, Xamarin offers a number of flavors: Xamarin.iOS and Xamarin.Mac, libraries that wrap native Apple APIs you can use to build apps for iOS and macOS using C# and Visual Studio; Xamarin.Android, a library that wraps native Java and Google APIs you can use to build apps for Android using C# and Visual Studio; and Xamarin.Forms, an open-source library that allows you to share code across platforms and build apps that run on Android, iOS, and Windows from a single C# codebase.

The biggest benefit of Xamarin.Forms is that you write code once, and it will run on all the supported platforms at no additional cost. As you'll learn throughout this ebook, Xamarin.Forms consists of a layer that wraps objects common to all the supported platforms into C# objects. Accessing native, platform-specific objects and APIs is possible in several ways (all discussed in the next chapters), but it requires some extra work. Additionally, Xamarin integrates with the Visual Studio IDE on Windows and is part of Visual Studio for Mac, so you can not only create cross-platform solutions, but also write code on different systems. If you look at Code Listing 1, you can see an example of XAML code that defines a page, and associated C# code that can define the behavior of the page.

*Code Listing 1*

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  x:Class="HelloWorld.HomePage">
</ContentPage>

using Xamarin.Forms;
namespace HelloWorld
{
  public partial class HomePage : ContentPage
  {
    public HomePage ()
    {
      InitializeComponent ();
    }
  }
}
```

The companion environment for Xamarin is the Microsoft App Center, a complete cloud solution for the app-management lifecycle that includes everything you need for build automation, internal distribution, analytics with diagnostics, and automated tests on more than 400 device configurations.

This book focuses on Xamarin.Forms and targets Visual Studio 2019 on Windows 10, but all the technical concepts apply to Visual Studio for Mac as well. However, if you prefer working on a Mac, I recommend that you read my book *Xamarin.Forms for macOS Succinctly*. I've also recorded a video series for Syncfusion that provides an overview of what Xamarin offers, and of Xamarin.iOS, Xamarin.Android, and Xamarin.Forms.

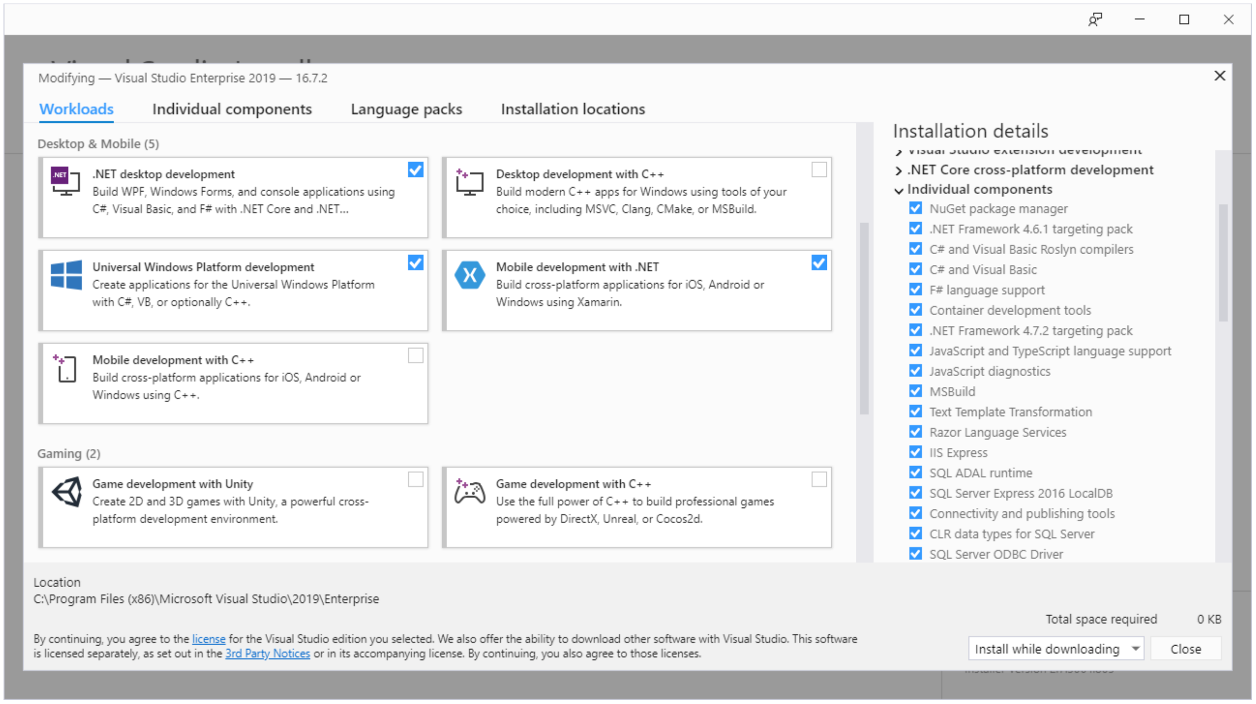
## Supported platforms

Out of the box, Xamarin.Forms allows creating apps for Android, iOS, and the Universal Windows Platform from a single C# codebase. Recently, the range of supported platforms has been expanded to include Tizen (an operating system by Samsung for proprietary devices). Additionally, Microsoft is working on supporting macOS, WPF, and GTK#. As you can imagine, this opens incredible opportunities in the market of cross-platform development because you can target both mobile and desktop systems. In this book, I will target the most popular operating systems (Android, iOS, and Windows 10) because support for other platforms is not yet released (apart from Tizen). You can read the official documentation about targeting macOS, Tizen, WPF, and GTK# in your Xamarin.Forms projects, with the assumption that your shared code will not change.

## Setting up the development environment

In order to build native mobile apps with Xamarin.Forms, you need Windows 10 as your operating system, and Microsoft Visual Studio 2019 as your development environment. You can download and install the Visual Studio 2019 Community edition for free and get all the necessary tools for Xamarin development. I will discuss the latest stable release of Xamarin.Forms, version 4.8, in this book, so make sure you install version 16.7.2 or later of Visual Studio 2019.

When you start the installation, you will need to select the **Mobile development with .NET** workload in the Visual Studio Installer (see Figure 1).



Modifying — Visual Studio Enterprise 2019 — 16.7.2

Workloads | Individual components | Language packs | Installation locations

Desktop & Mobile (5)

- [✓] .NET desktop development — Build WPF, Windows Forms, and console applications using C#, Visual Basic, and F# with .NET Core and .NET...
- [ ] Desktop development with C++ — Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.
- [✓] Universal Windows Platform development — Create applications for the Universal Windows Platform with C#, VB, or optionally C++.
- [✓] Mobile development with .NET — Build cross-platform applications for iOS, Android or Windows using Xamarin.
- [ ] Mobile development with C++ — Build cross-platform applications for iOS, Android or Windows using C++.

Gaming (2)

- [ ] Game development with Unity — Create 2D and 3D games with Unity, a powerful cross-platform development environment.
- [ ] Game development with C++ — Use the full power of C++ to build professional games powered by DirectX, Unreal, or Cocos2d.

Installation details

- .NET Core cross-platform development
- Individual components
  - [✓] NuGet package manager
  - [✓] .NET Framework 4.6.1 targeting pack
  - [✓] C# and Visual Basic Roslyn compilers
  - [✓] C# and Visual Basic
  - [✓] F# language support
  - [✓] Container development tools
  - [✓] .NET Framework 4.7.2 targeting pack
  - [✓] JavaScript and TypeScript language support
  - [✓] JavaScript diagnostics
  - [✓] MSBuild
  - [✓] Text Template Transformation
  - [✓] Razor Language Services
  - [✓] IIS Express
  - [✓] SQL ADAL runtime
  - [✓] SQL Server Express 2016 LocalDB
  - [✓] Connectivity and publishing tools
  - [✓] CLR data types for SQL Server
  - [✓] SQL Server ODBC Driver

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise

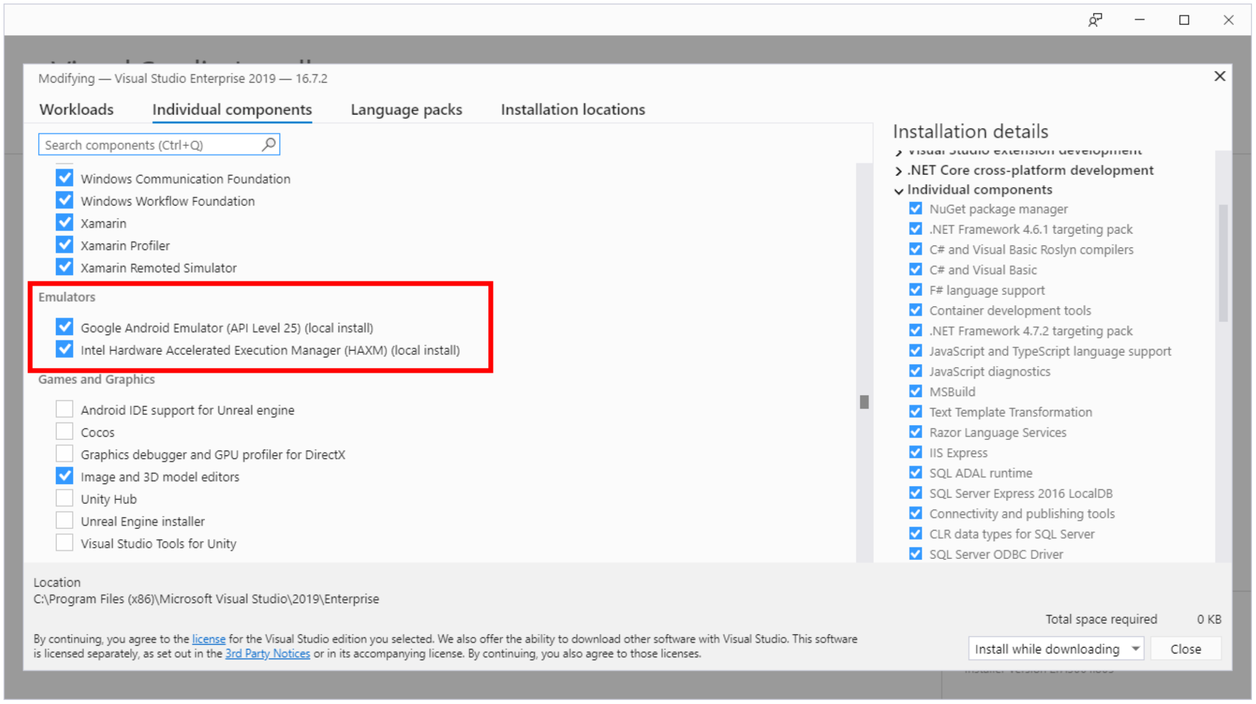
Total space required 0 KB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

*Figure 1: Installing Xamarin development tools*

When you select this workload, the Visual Studio Installer will download and install all the necessary tools to build apps for Android, iOS, and Windows. iOS requires additional configuration, described in the next section.

For Windows 10 development, you need additional tools and SDKs, which you can get by also selecting the **Universal Windows Platform development** workload. If you select the **Individual components** tab, you will have an option to check if Android and Windows emulators have been selected or to make a choice manually (see Figure 2).


Modifying — Visual Studio Enterprise 2019 — 16.7.2

Workloads | Individual components | Language packs | Installation locations

Search components (Ctrl+Q)

- [✓] Windows Communication Foundation
- [✓] Windows Workflow Foundation
- [✓] Xamarin
- [✓] Xamarin Profiler
- [✓] Xamarin Remoted Simulator

Emulators

- [✓] Google Android Emulator (API Level 25) (local install)
- [✓] Intel Hardware Accelerated Execution Manager (HAXM) (local install)

Games and Graphics

- [ ] Android IDE support for Unreal engine
- [ ] Cocos
- [ ] Graphics debugger and GPU profiler for DirectX
- [✓] Image and 3D model editors
- [ ] Unity Hub
- [ ] Unreal Engine installer
- [ ] Visual Studio Tools for Unity

Installation details

› .NET Core cross-platform development

⌄ Individual components

- [✓] NuGet package manager
- [✓] .NET Framework 4.6.1 targeting pack
- [✓] C# and Visual Basic Roslyn compilers
- [✓] C# and Visual Basic
- [✓] F# language support
- [✓] Container development tools
- [✓] .NET Framework 4.7.2 targeting pack
- [✓] JavaScript and TypeScript language support
- [✓] JavaScript diagnostics
- [✓] MSBuild
- [✓] Text Template Transformation
- [✓] Razor Language Services
- [✓] IIS Express
- [✓] SQL ADAL runtime
- [✓] SQL Server Express 2016 LocalDB
- [✓] Connectivity and publishing tools
- [✓] CLR data types for SQL Server
- [✓] SQL Server ODBC Driver

Location
C:\Program Files (x86)\Microsoft Visual Studio\2019\Enterprise

Total space required 0 KB

By continuing, you agree to the license for the Visual Studio edition you selected. We also offer the ability to download other software with Visual Studio. This software is licensed separately, as set out in the 3rd Party Notices or in its accompanying license. By continuing, you also agree to those licenses.

Install while downloading | Close

*Figure 2: Selecting emulators*

Whether you will use Visual Studio 2019, Visual Studio for Mac, or both, I suggest you install the Google emulator, which has an identical appearance and behavior on both systems.

For the Windows 10 emulator, my suggestion is to download the oldest version if you plan to target older versions of Windows 10; otherwise, the most recent is always a good option. Go ahead with the installation and wait for it to complete.

## Configuring a Mac

Apple's policies establish that a Mac computer is required to build an app. This is because only the Xcode development environment and the Apple SDKs are allowed to run the build process. A Mac computer is also needed for code signing, setting up profiles, and publishing an app to the App Store. You can use a local Mac in your network, which also allows you to debug and test apps on a physical device, or a remote Mac. In both cases, macOS must be configured with the following software requirements:

- macOS "High Sierra" (10.13) or higher.
- Xcode and Apple SDKs, which you get from the App Store for free.
- Xamarin.iOS engine. The easiest way to get Xamarin properly configured on a Mac is by installing Visual Studio Community for Mac.

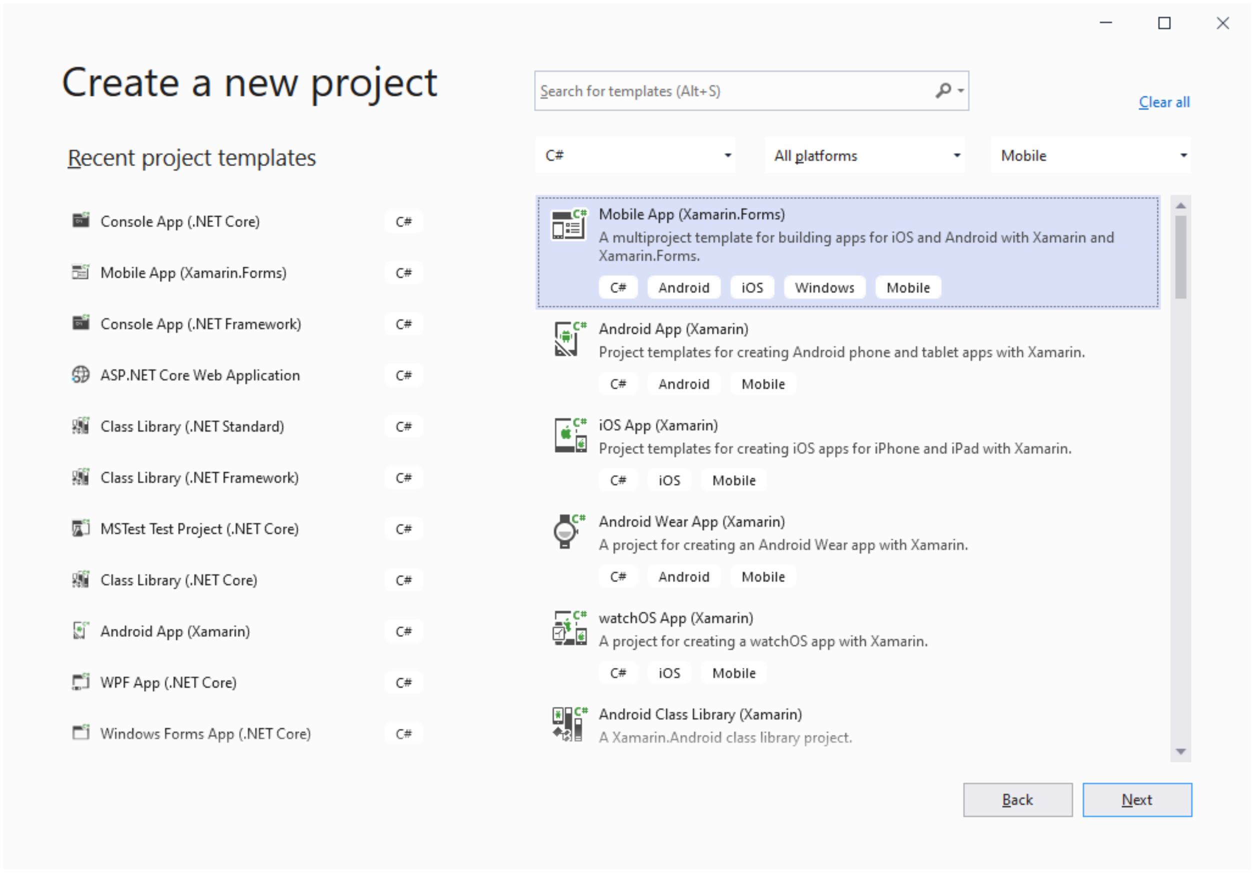
Visual Studio will connect to the Mac to launch the Xcode compiler and SDKs; remote connections must be enabled for the latter. The official Xamarin documentation has a specific page that will help you configure a Mac. I recommend you read it carefully, especially because it explains how to configure profiles and certificates, and how to use Xcode to perform preliminary configurations. The documentation is actually about Xamarin.iOS, but the same steps apply to Xamarin.Forms.

***Tip: Visual Studio 2019 includes integrated tools that simplify the Mac configuration from within the IDE, and without the need to work on the Mac directly. These improvements are discussed in the Xamarin.iOS project section later in this chapter.***

## Creating Xamarin.Forms solutions

Assuming that you have installed and configured your development environment, the next step is opening Visual Studio to see how you create Xamarin.Forms solutions, and what these solutions consist of.

In the Start window, click **Create new project**. When the **Create a new project** dialog appears, apply the following filters under the search box: **C#**, **All Platforms**, and **Mobile**. This will restrict the list of project templates to Xamarin only. For Xamarin.Forms, the project templates of interest are **Mobile App (Xamarin.Forms)**, which is the first in the list (see Figure 3), and **Xamarin.UITest Cross-Platform Test Project**, which is the last one in the list, and that you will see by scrolling to the bottom.


Create a new project

Search for templates (Alt+S)

Clear all

Recent project templates

| | |
|---|---|
| Console App (.NET Core) | C# |
| Mobile App (Xamarin.Forms) | C# |
| Console App (.NET Framework) | C# |
| ASP.NET Core Web Application | C# |
| Class Library (.NET Standard) | C# |
| Class Library (.NET Framework) | C# |
| MSTest Test Project (.NET Core) | C# |
| Class Library (.NET Core) | C# |
| Android App (Xamarin) | C# |
| WPF App (.NET Core) | C# |
| Windows Forms App (.NET Core) | C# |

C# | All platforms | Mobile

Mobile App (Xamarin.Forms)
A multiproject template for building apps for iOS and Android with Xamarin and Xamarin.Forms.
C# Android iOS Windows Mobile

Android App (Xamarin)
Project templates for creating Android phone and tablet apps with Xamarin.
C# Android Mobile

iOS App (Xamarin)
Project templates for creating iOS apps for iPhone and iPad with Xamarin.
C# iOS Mobile

Android Wear App (Xamarin)
A project for creating an Android Wear app with Xamarin.
C# Android Mobile

watchOS App (Xamarin)
A project for creating a watchOS app with Xamarin.
C# iOS Mobile

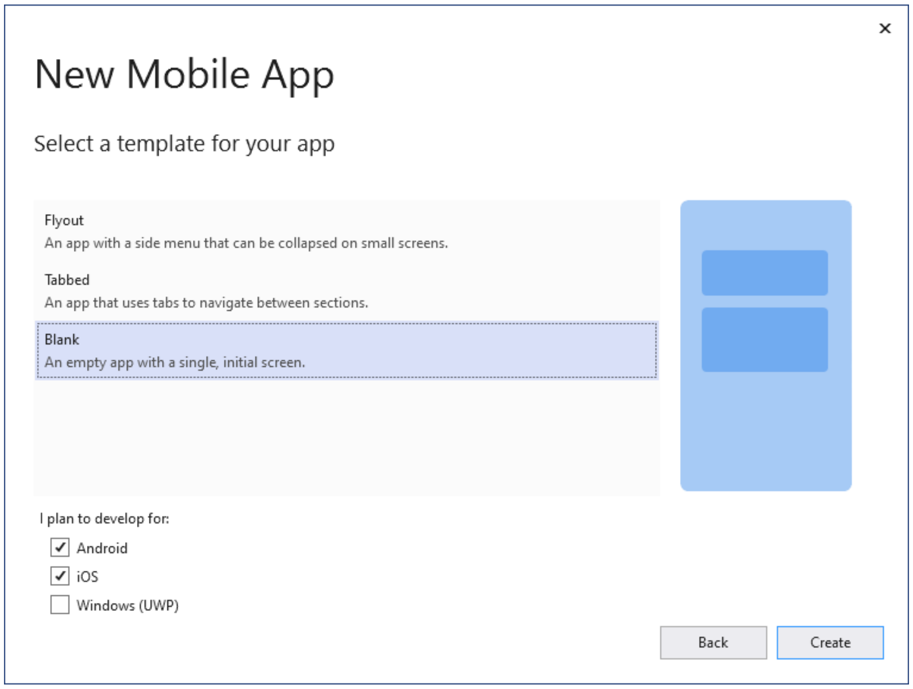
Android Class Library (Xamarin)
A Xamarin.Android class library project.

*Figure 3: Project templates for Xamarin.Forms*

The **Mobile App (Xamarin.Forms)** template is the one you use to build mobile apps. The **Xamarin.UITest Cross-Platform Test Project** template is used to create automated UI tests, but this will not be discussed in this book. Additionally, Visual Studio 2019 offers the Class Library (.NET Standard) project template, which can be used to create reusable class libraries that a Xamarin.Forms app can consume. More information on .NET Standard will come in Chapter 2.

Select the **Mobile App (Xamarin.Forms)** template and click **Next**. In the next screen, Visual Studio will ask you to specify a project name and a location. Provide a name of your choice or leave the default name and click **Create**.

At this point, Visual Studio will ask you to select from the Flyout, Tabbed, and Blank templates (see Figure 4). The Flyout template generates a basic user interface based on a common infrastructure called Shell (see Chapter 6), pages and visual elements that will be discussed later, with some sample data. The Tabbed template generates a basic user interface based on tabs to navigate among child pages. Neither Flyout nor Tabbed are a good starting point (unless you already have experience with Xamarin); moreover, both templates rely on the Shell component, which is not available on Universal Windows Platform for now, so select the **Blank** template.


New Mobile App

Select a template for your app

Flyout
An app with a side menu that can be collapsed on small screens.

Tabbed
An app that uses tabs to navigate between sections.

Blank
An empty app with a single, initial screen.

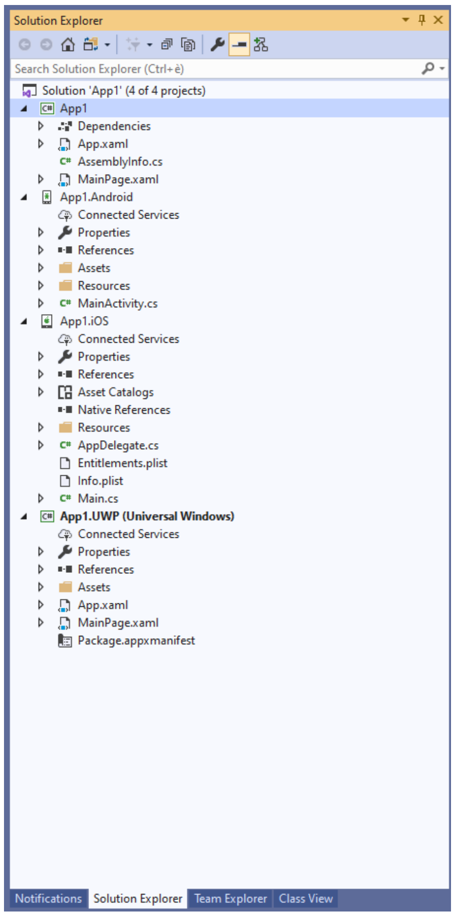
I plan to develop for:

- [x] Android
- [x] iOS
- [ ] Windows (UWP)

*Figure 4: Selecting the template, UI technology, and code sharing strategy for a new project*

In the **I plan to develop for:** group, select all the platforms you want to target. The default is Android and iOS, but you can also select Windows (UWP) with the Blank template.

In Solution Explorer, you will see that the solution is made up of four projects, as demonstrated in Figure 5.



```
Solution 'App1' (4 of 4 projects)
  App1
    Dependencies
    App.xaml
    AssemblyInfo.cs
    MainPage.xaml
  App1.Android
    Connected Services
    Properties
    References
    Assets
    Resources
    MainActivity.cs
  App1.iOS
    Connected Services
    Properties
    References
    Asset Catalogs
    Native References
    Resources
    AppDelegate.cs
    Entitlements.plist
    Info.plist
    Main.cs
  App1.UWP (Universal Windows)
    Connected Services
    Properties
    References
    Assets
    App.xaml
    MainPage.xaml
    Package.appxmanifest
```

*Figure 5: The structure of a Xamarin.Forms solution*

The first project is a .NET Standard library. This project contains all the code that can be shared across platforms, and its implementation will be discussed in the next chapter. For now, what you need to know is that this project is the place where you will write all the user interface of your app, and all the code that does not require interacting with native APIs.

The second project, whose suffix is **Android**, is a Xamarin.Android native project. It has a reference to the shared code and to Xamarin.Forms, and it implements the required infrastructure for your app to run on Android devices.

The third project, whose suffix is **iOS**, is a Xamarin.iOS native project. This one also has a reference to the shared code and to Xamarin.Forms, and it implements the required infrastructure for your app to run on the iPhone and iPad.

The fourth and last project is a native Universal Windows project (UWP) that has a reference to shared code and implements the infrastructure for your app to run on Windows 10 devices, for both the desktop and mobile devices. I will now provide more details on each platform project, so that you have a basic knowledge of their properties. This is very important, because you will need to fine-tune project properties every time you create a new Xamarin.Forms solution.

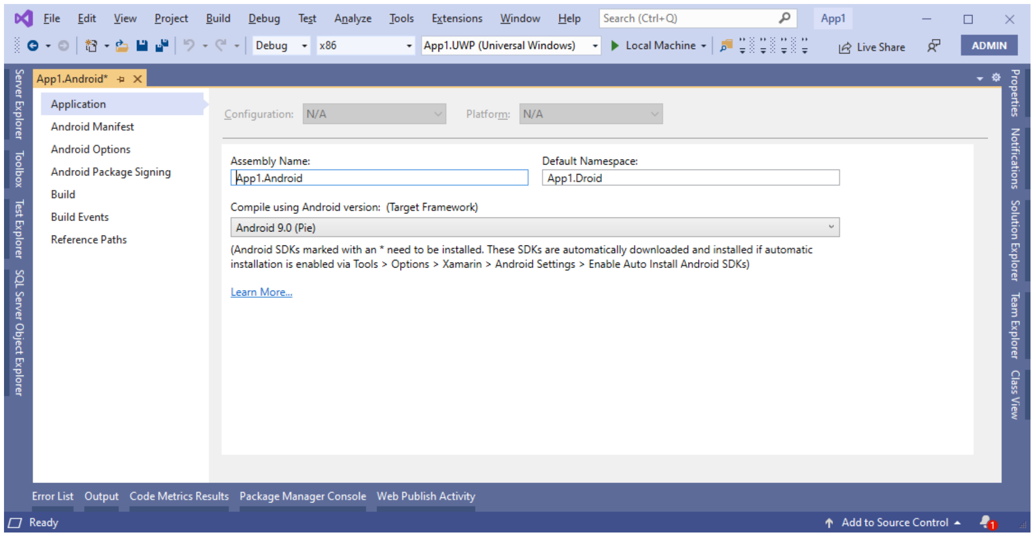
## The Xamarin.Forms library

Technically speaking, Xamarin.Forms is a .NET library that exposes all the objects discussed in this book through a root namespace called `Xamarin.Forms`. It has been recently open sourced, and it ships as a NuGet package that Visual Studio automatically installs in all projects when you create a new solution. The NuGet Package Manager in Visual Studio will then notify you of available updates. Because creating Xamarin.Forms solutions should be allowed even if your PC is offline, Visual Studio installs the version of the NuGet package in the local cache, which isn't typically the latest version available. For this reason, it is recommended that you upgrade the Xamarin.Forms and other Xamarin packages in the solution to the latest version, and that you only upgrade to stable releases. Though alpha and beta intermediate releases are often available, their only intended use is for experimenting with new features still in development. As of this writing, version 5.0.0.1931 is the latest stable release, and is required to successfully complete all the exercises and code examples provided in the next chapters.

## The Xamarin.Android project

Xamarin.Android makes it possible for your Xamarin.Forms solution to run on Android devices. The **MainActivity.cs** file represents the startup activity for the Android app that Xamarin generates. In Android, an activity can be thought of as a single screen with a user interface, and every app has at least one. In this file, Visual Studio adds startup code that you should not change, especially the initialization code for Xamarin.Forms you see in the last two lines of code. In this project, you can add code that requires accessing native APIs and platform-specific features, as you will learn in Chapter 9.

The **Resources** folder is also very important, because it contains subfolders where you can add icons and images for different screen resolutions. The name of such folders starts with **drawable**, and each represents a particular screen resolution. The Xamarin documentation explains thoroughly how to provide icons and images for different resolutions on Android. The **Properties** element in Solution Explorer allows you to access the project properties, as you would do with any C# solution. In the case of Xamarin, in the **Application** tab (see Figure 6) you can specify the version of the Android SDK that Visual Studio should use to build the app package.



Configuration: N/A Platform: N/A

Assembly Name: App1.Android

Default Namespace: App1.Droid

Compile using Android version: (Target Framework)

Android 9.0 (Pie)

(Android SDKs marked with an * need to be installed. These SDKs are automatically downloaded and installed if automatic installation is enabled via Tools > Options > Xamarin > Android Settings > Enable Auto Install Android SDKs)

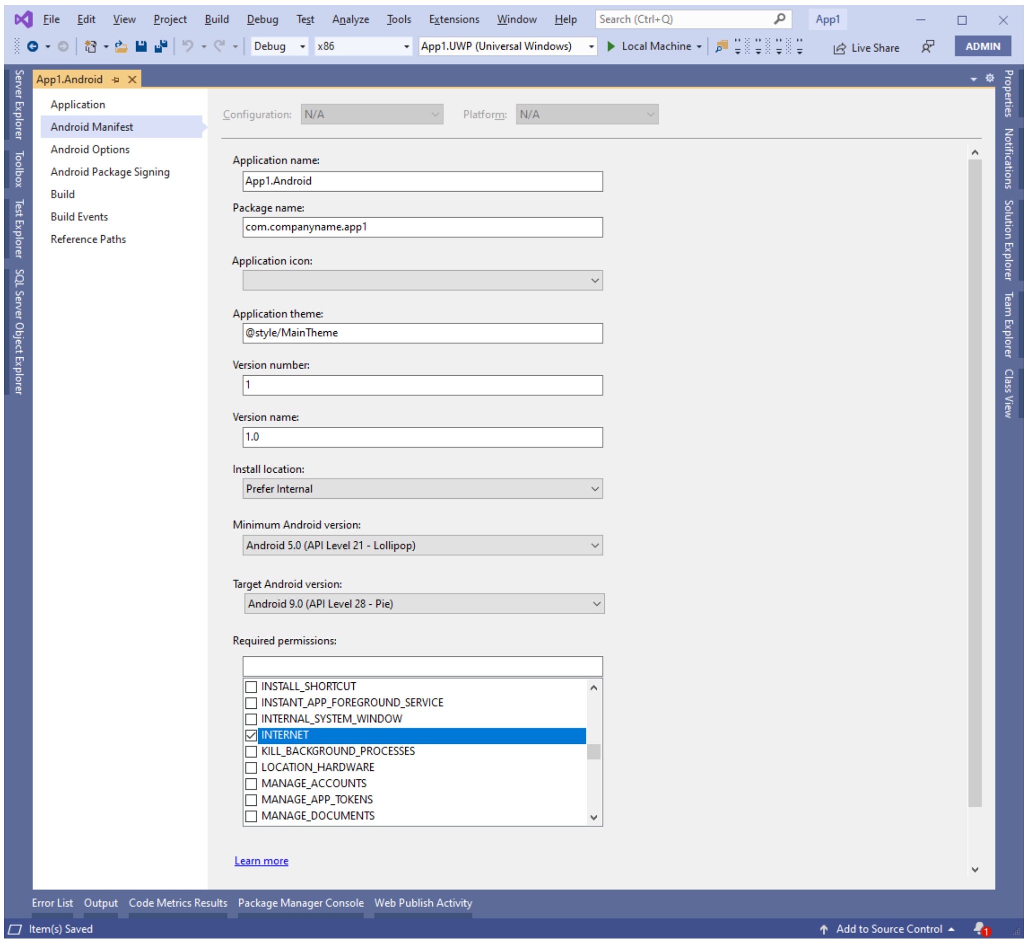
Learn More...

*Figure 6: Selecting the version of the Android SDK for compilation*

Visual Studio automatically selects the latest version available and provides a drop-down list where you can select a different version of the SDK. Notice that Visual Studio marks an SDK version with an * symbol if it is not installed on the development machine. You can download the SDK using the Android SDK Manager tool, or you can enable automatic download via **Tools** > **Options** > **Xamarin** > **Android Settings** > **Auto Install Android SDKs**. The SDK selection here does not affect the minimum version of Android you want to target; instead, it is related to the version of the build tools that Visual Studio will use. My recommendation is to leave the default selection unchanged.

***Tip: You can manage installed SDK versions using the Android SDK Manager, a tool that you can launch from both the Windows Programs menu and from Visual Studio by selecting*** *Tools* ***>*** *Android* ***>*** *Android SDK Manager****.***

The **Android Manifest** tab is even more important. Here you specify your app's metadata, such as name, version number, icon, and permissions the user must grant to the application. Figure 7 shows an example.


File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
App1
Debug
x86
App1.UWP (Universal Windows)
Local Machine
Live Share
ADMIN
App1.Android
Server Explorer Toolbox Test Explorer SQL Server Object Explorer
Properties Notifications Solution Explorer Team Explorer Class View
Application
Android Manifest
Android Options
Android Package Signing
Build
Build Events
Reference Paths
Learn more
Error List Output Code Metrics Results Package Manager Console Web Publish Activity
Item(s) Saved
Add to Source Control

Configuration: N/A Platform: N/A

Application name: App1.Android

Package name: com.companyname.app1

Application icon:

Application theme: @style/MainTheme

Version number: 1

Version name: 1.0

Install location: Prefer Internal

Minimum Android version: Android 5.0 (API Level 21 - Lollipop)

Target Android version: Android 9.0 (API Level 28 - Pie)

Required permissions:

- [ ] INSTALL_SHORTCUT
- [ ] INSTANT_APP_FOREGROUND_SERVICE
- [ ] INTERNAL_SYSTEM_WINDOW
- [x] INTERNET
- [ ] KILL_BACKGROUND_PROCESSES
- [ ] LOCATION_HARDWARE
- [ ] MANAGE_ACCOUNTS
- [ ] MANAGE_APP_TOKENS
- [ ] MANAGE_DOCUMENTS

*Figure 7: The Android manifest*

The information you supply in the Android Manifest tab is also important for publication to Google Play. For example, the package name uniquely identifies your app package in the Google Play store; by convention, it is in the following form: com.*companyname.appname*, which is self-explanatory (com. is a conventional prefix). The version name is your app version, whereas the version number is a single-digit string that represents updates. For instance, you might have version name 1.0 and version number 1, version name 1.1 and version number 2, version name 1.2 and version number 3, and so on.

The **Install location** option allows you to specify whether your app should be installed only in the internal storage or if memory cards are allowed, but remember that starting from Android 6.0, apps can no longer be installed onto a removable storage device. In the **Minimum Android version** drop-down list, you can select the minimum Android version you want to target.

It is important that you pay particular attention to the **Required permissions** list. Here you must specify all the permissions that your app must be granted in order to access resources such as the internet, the camera, other hardware devices, sensors, and more. Remember that, starting from Android 6.0, the operating system will ask the user for confirmation before accessing a resource that requires one of the permissions you marked in the manifest, and the app will fail if it attempts to access a sensitive resource, but the related permission was not selected in the manifest.

In the **Android Options** tab, you will be able to manage debugging and build options. However, I will not walk through all the available options here. It is worth highlighting the **Use Fast Deployment** option though, which is enabled by default. When enabled, deploying the app to a physical or emulated device will only replace changed files. This can often cause the app to not work properly or not start at all, so my suggestion is you disable this option. The other tabs are the same as for other .NET projects.

## The Xamarin.iOS project

Similar to the Xamarin.Android project, the Xamarin.iOS project makes it possible for your Xamarin.Forms solutions to run on the iPhone and iPad. Supposing you have a configured Mac, Visual Studio will need to know its address in the network. Visual Studio normally asks this after creating a new Xamarin.Forms solution or opening an existing one, but you can manually enter the Mac address by choosing **Tools** > **iOS** > **Pair to Mac**. In the **Pair to Mac** dialog window, Visual Studio should be able to list any detected Mac computers in the network. However, it is strongly recommended that you re-add a Mac by providing its IP address rather than its name. For example, Figure 8 shows the **Pair to Mac** dialog window displaying my MacBook Pro machine with both its name and its IP address, but Visual Studio establishes a connection based on the IP, not the name.

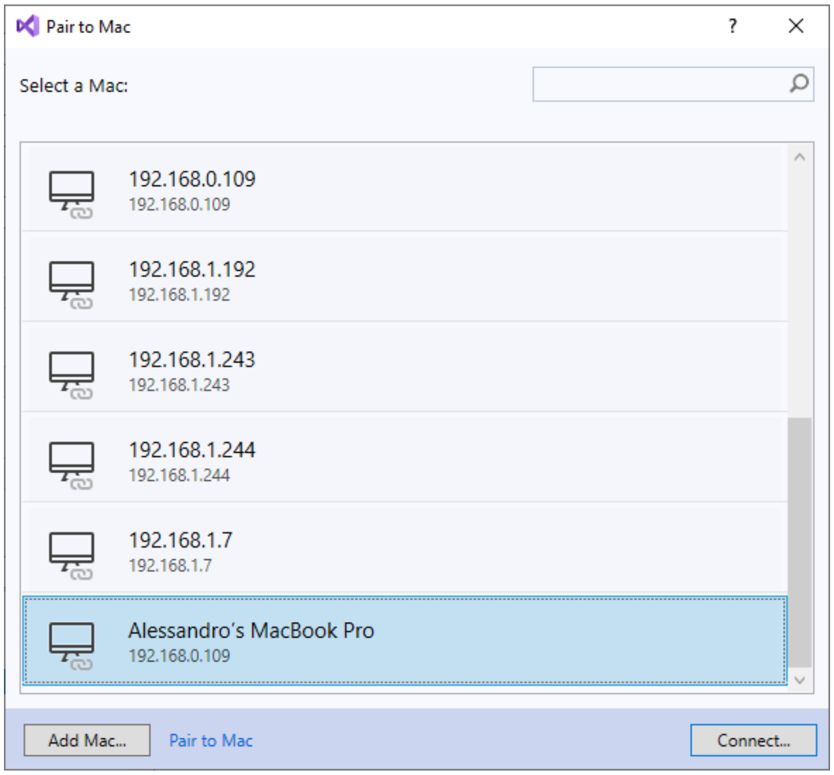
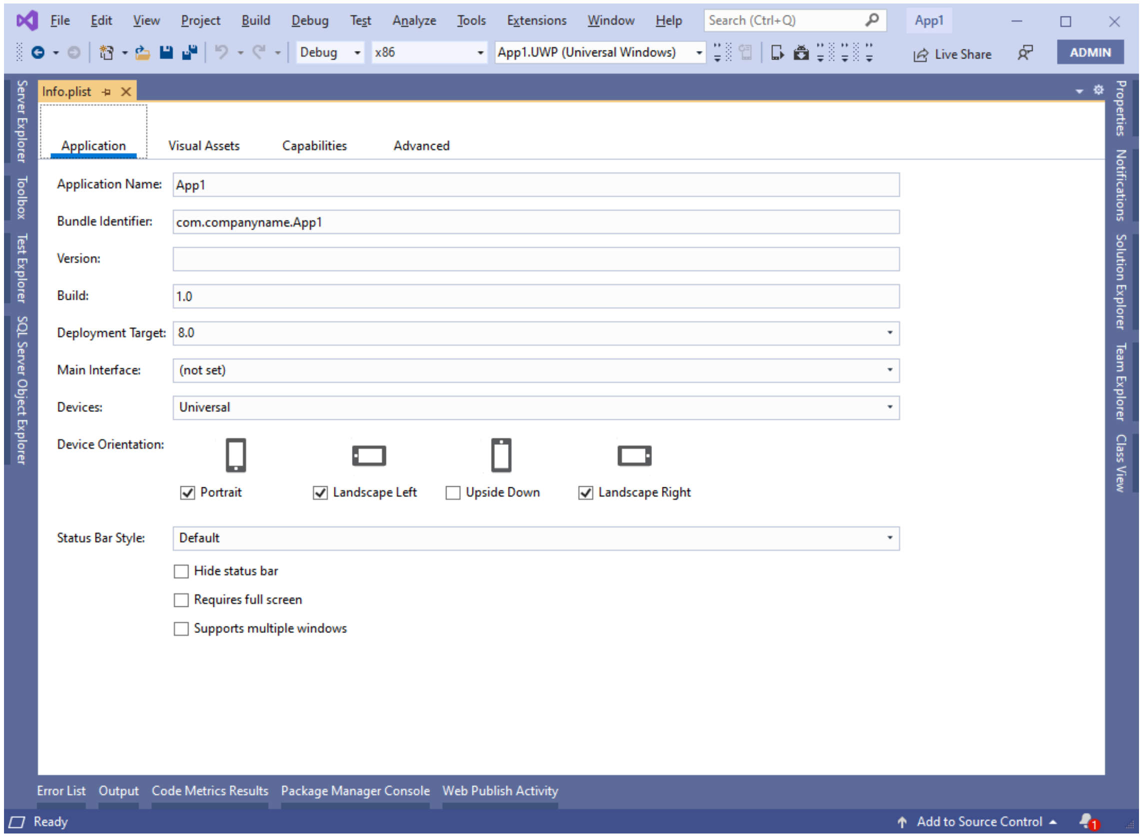


*Figure 8: Connecting Visual Studio to a Mac*

If a Mac is not detected, click **Add Mac** and enter its IP address first. Then, when requested, enter the same credentials you use to log in to the Mac. If the connection succeeds, Visual Studio will show a success message in the status bar.

For the Xamarin.iOS project, the **AppDelegate.cs** file contains the Xamarin.Forms initialization code and should not be changed. You can add all the code that requires accessing native APIs and platform-specific features in this project, as you will learn in Chapter 9. In the **Info.plist** (property list) file (see Figure 9) and, through each tab, you can configure your app metadata, the minimum target version, supported devices and orientations, capabilities (such as Game Center and Maps integration), visual assets (such as launch images and icons), and other advanced features.

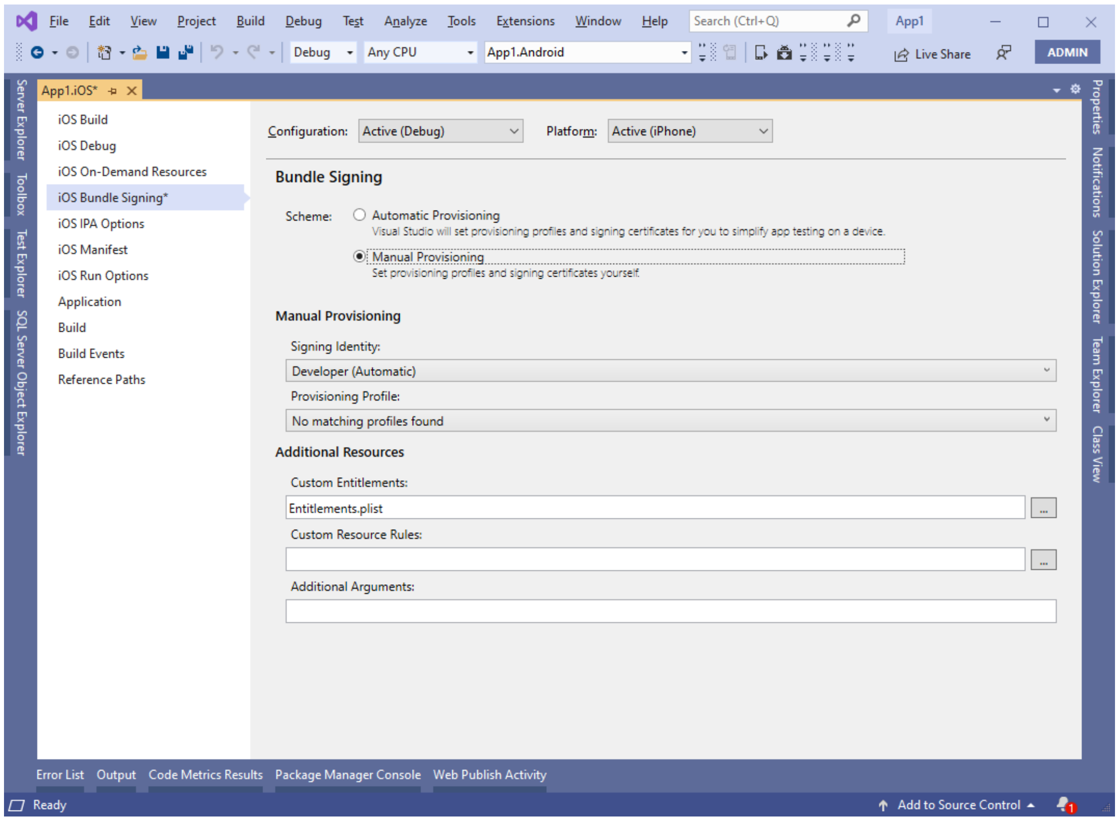


| | |
|---|---|
| Application Name: | App1 |
| Bundle Identifier: | com.companyname.App1 |
| Version: | |
| Build: | 1.0 |
| Deployment Target: | 8.0 |
| Main Interface: | (not set) |
| Devices: | Universal |
| Device Orientation: | ☑ Portrait ☑ Landscape Left ☐ Upside Down ☑ Landscape Right |
| Status Bar Style: | Default |
| | ☐ Hide status bar |
| | ☐ Requires full screen |
| | ☐ Supports multiple windows |

*Figure 9: The Info.plist file*

The **Info.plist** file represents the app manifest in iOS, and therefore is not related to only Xamarin.iOS. In fact, if you have experience with Xcode and native iOS development, you already know this file. Unlike Android, the iOS operating system includes restriction policies that are automatically applied to most sensitive resources, especially those involving security and privacy. Also, there are differences among iOS 8.x, 9.x, 10.x, and 11.x in how the OS handles these options.

The Info.plist reference will help you understand how to properly configure any exceptions. Among the project properties, the most important is, without a doubt, the iOS Bundle Signing. You use the iOS Bundle Signing properties to specify the identity that the Apple tools must use to sign the app package, and to specify the provisioning profile that is used to associate a team of developers to an app identifier. Configuring signing identities and profiles is particularly important when preparing an app for publishing. Figure 10 shows the iOS Bundle Signing properties.



Configuration: Active (Debug)  Platform: Active (iPhone)

Bundle Signing

Scheme: Automatic Provisioning [✓ Manual Provisioning]

Automatic Provisioning: Visual Studio will set provisioning profiles and signing certificates for you to simplify app testing on a device.

Manual Provisioning: Set provisioning profiles and signing certificates yourself.

Manual Provisioning

Signing Identity: Developer (Automatic)

Provisioning Profile: No matching profiles found

Additional Resources

Custom Entitlements: Entitlements.plist

Custom Resource Rules:

Additional Arguments:

*Figure 10: The iOS Bundle Signing options*

With the Automatic Provisioning option, you will just need to enter the Apple ID associated to an active Apple Developer account and Visual Studio will generate all the provisioning profiles for you. The next section discusses this in more detail. With the Manual Provisioning option, you will be able to adjust your settings manually.

As you can imagine, Visual Studio can detect available signing identities and provisioning profiles only when connected to a Mac, because this information is generated via Xcode. Further details about configuring a Xamarin.iOS project are offered through the documentation.

## Automatic iOS provisioning

Whether you use native Xamarin.iOS or Xamarin.Forms, creating apps for iOS requires some preliminary steps that can be really complicated and frustrating. In fact, apart from enrolling in Apple's Developer Program, which is required to enable your Apple developer account to sign and publish apps to the App Store, you need to provision your iOS device. iOS provisioning will enable you to deploy and test your apps on physical devices.

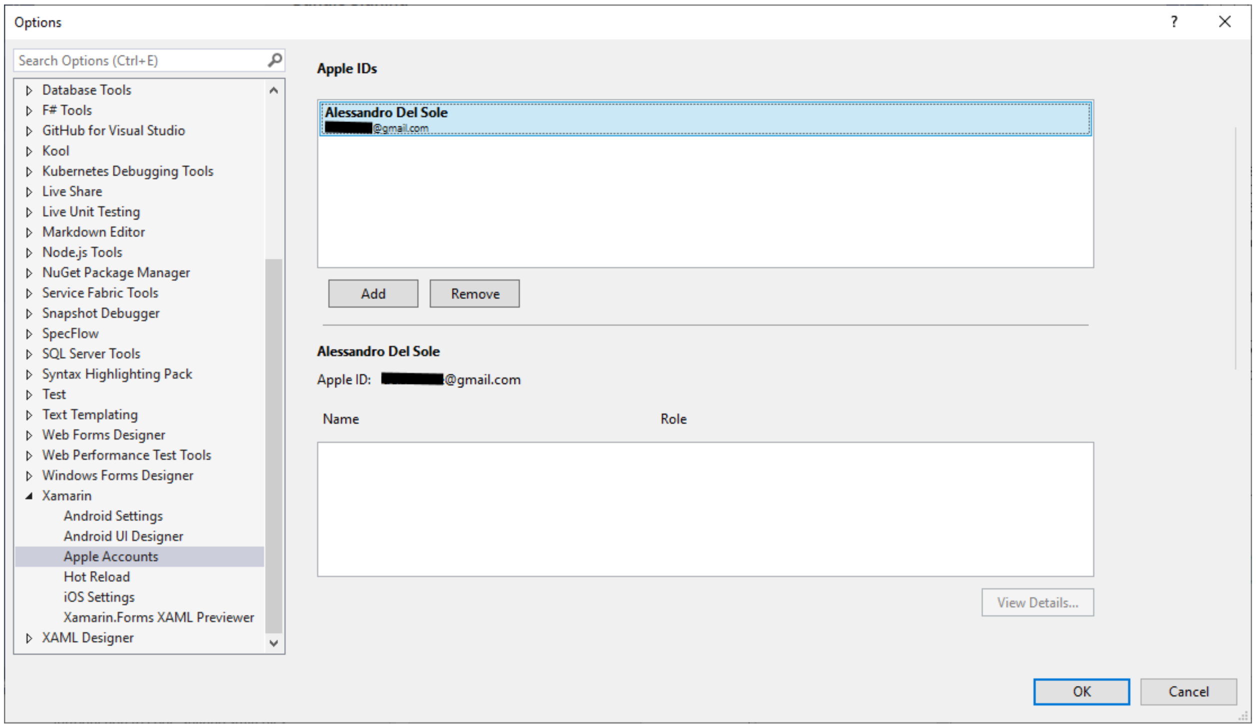
iOS provisioning is usually done via Xcode on the Mac and requires you to:

- **Create a development team**: This includes specifying the list of Apple developer accounts in the development team, and enabling them to sign and publish an app.

- **Set up provisioning profiles**: A provisioning profile is bundled into the compiled app package and contains three types of information: a unique app identifier (App ID), one or more development certificates required to test an app on physical devices, and a Unique Device Identifiers (UDI) list that enumerates devices allowed to run an app.
- **Create signing certificates**: All apps must be signed before they can run on an iOS device, even for development, so a signing certificate is required. Different kinds of certificates are available to each developer account (such as development and publishing), depending on the level of subscription.

Even though the documentation nicely explains how to get started with iOS provisioning, the reality is that complexity is the biggest barrier for developers wanting to build apps for iOS. Fortunately, Visual Studio 2019 includes support for automatic iOS provisioning. You simply provide your Apple developer account, and Visual Studio 2019 will set up all the necessary artifacts on your behalf through a connection to a Mac machine.

To accomplish this, you must first associate your Apple ID with Visual Studio by selecting **Tools** > **Options** > **Xamarin** > **Apple Accounts**. In the **Options** dialog box, you'll be able to associate one or more Apple IDs by clicking **Add**, after which Visual Studio 2019 shows the list of associated teams and the user role for each team (see Figure 11).



Alessandro Del Sole
[redacted]@gmail.com

Alessandro Del Sole
Apple ID: [redacted]@gmail.com

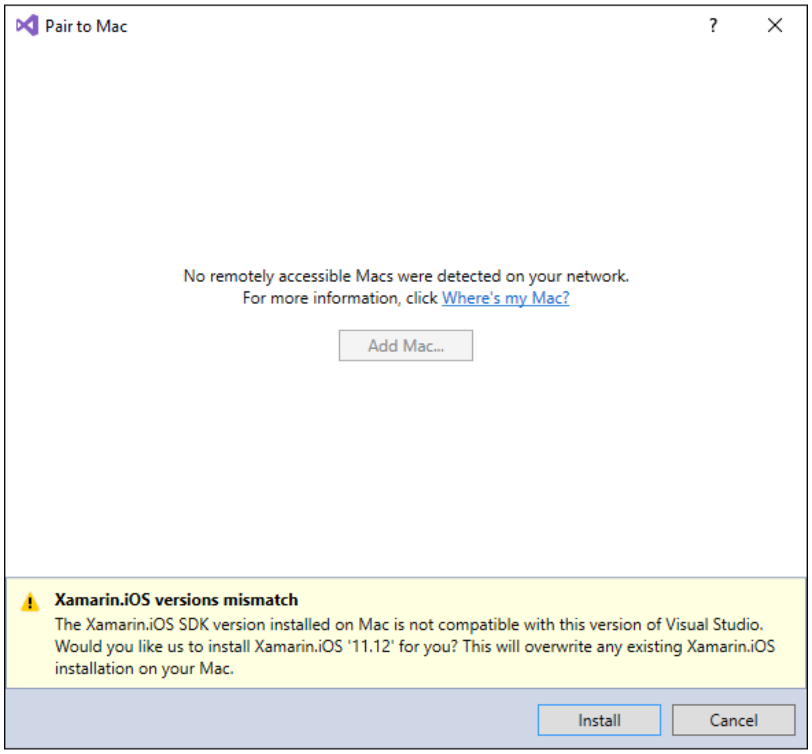
*Figure 11: Associating development teams with Visual Studio*

Once you click **OK**, you can go to the iOS Bundle Signing tab of the Xamarin.iOS project properties and select the **Automatic Provisioning** option (see Figure 10). You'll simply need to select the team you wish to use for development from the **Team** drop-down menu, and Visual Studio will generate the necessary provisioning profiles and signing certificates required to test apps on a physical device (which must be connected to the Mac machine).

## Remote Xamarin.iOS update

One important thing that you must keep in mind when building iOS apps with Xamarin is that the same version of the SDK must be installed on both the PC and the Mac. In the past, you had to update the Xamarin.iOS SDK manually on both systems. Visual Studio 2019 has an integrated feature that will help you update the Xamarin.iOS version remotely, without the need to access the physical, remote Mac. When connecting to the Mac, if Visual Studio 2019 detects a version mismatch, it will show a warning message and provide an Install button that will launch the update process on the remote Mac.

Figure 12 shows the warning message and the Install button (you can click **Cancel** if you do not want to go through the update process).



No remotely accessible Macs were detected on your network.
For more information, click Where's my Mac?

**Xamarin.iOS versions mismatch**
The Xamarin.iOS SDK version installed on Mac is not compatible with this version of Visual Studio. Would you like us to install Xamarin.iOS '11.12' for you? This will overwrite any existing Xamarin.iOS installation on your Mac.

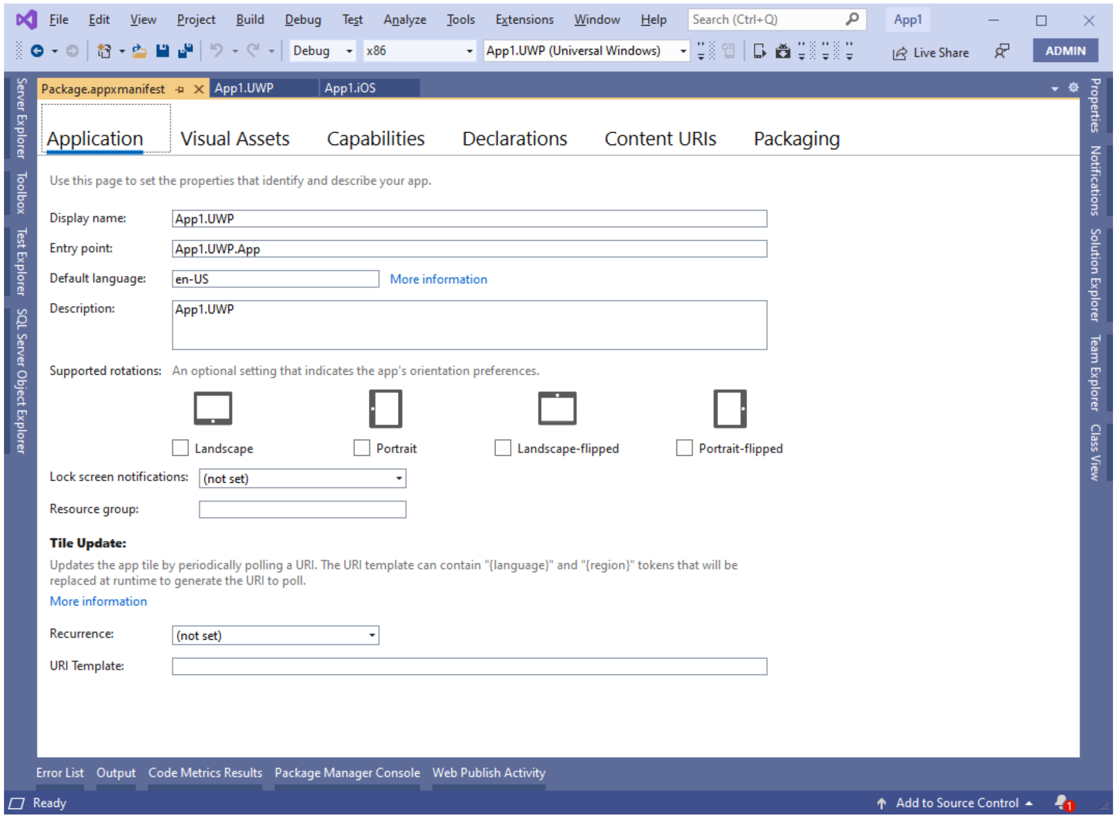
*Figure 12: Remote update of Xamarin.iOS*

## The Universal Windows Platform project

The Universal Windows Platform project in a Xamarin.Forms solution is a normal UWP project with a reference to the shared code and to the Xamarin.Forms package.

In the **App.xaml.cs** file, you can see initialization code, which you must not change. What you will need to configure, instead, is the application manifest, which you can edit by double-clicking the **Package.appxmanifest** file in Solution Explorer.

Visual Studio has a nice editor for UWP manifests, and you will at least configure app metadata (see Figure 13), visual assets such as icons and logos, and capabilities (see Figure 14). These include permissions you need to specify before testing and distributing your apps, and Windows 10 will ask the user for confirmation before accessing resources that require a permission.



Use this page to set the properties that identify and describe your app.

Display name: App1.UWP

Entry point: App1.UWP.App

Default language: en-US More information

Description: App1.UWP

Supported rotations: An optional setting that indicates the app's orientation preferences.

☐ Landscape ☐ Portrait ☐ Landscape-flipped ☐ Portrait-flipped

Lock screen notifications: (not set)

Resource group:

**Tile Update:**

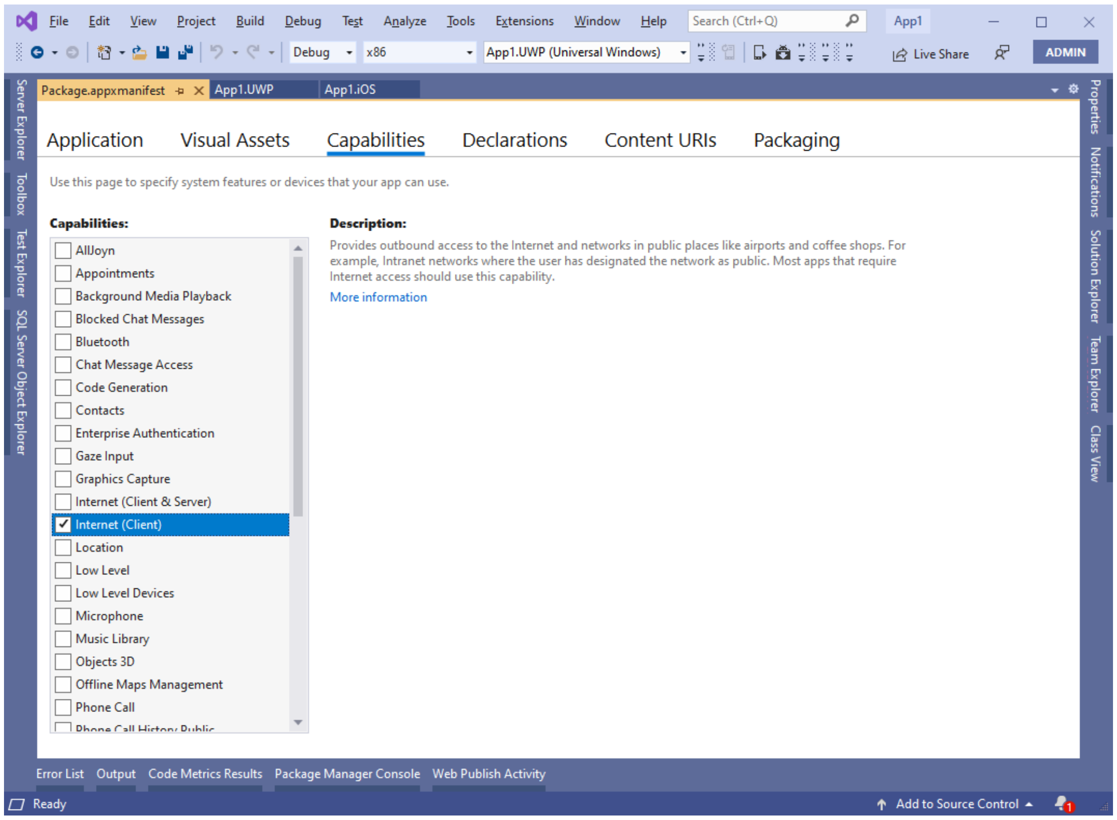
Updates the app tile by periodically polling a URI. The URI template can contain "{language}" and "{region}" tokens that will be replaced at runtime to generate the URI to poll.

More information

Recurrence: (not set)

URI Template:

*Figure 13: Editing metadata in UWP projects*



Use this page to specify system features or devices that your app can use.

**Capabilities:**

- [ ] AllJoyn
- [ ] Appointments
- [ ] Background Media Playback
- [ ] Blocked Chat Messages
- [ ] Bluetooth
- [ ] Chat Message Access
- [ ] Code Generation
- [ ] Contacts
- [ ] Enterprise Authentication
- [ ] Gaze Input
- [ ] Graphics Capture
- [ ] Internet (Client & Server)
- [✓] Internet (Client)
- [ ] Location
- [ ] Low Level
- [ ] Low Level Devices
- [ ] Microphone
- [ ] Music Library
- [ ] Objects 3D
- [ ] Offline Maps Management
- [ ] Phone Call
- [ ] Phone Call History Public

**Description:**

Provides outbound access to the Internet and networks in public places like airports and coffee shops. For example, Intranet networks where the user has designated the network as public. Most apps that require Internet access should use this capability.

More information

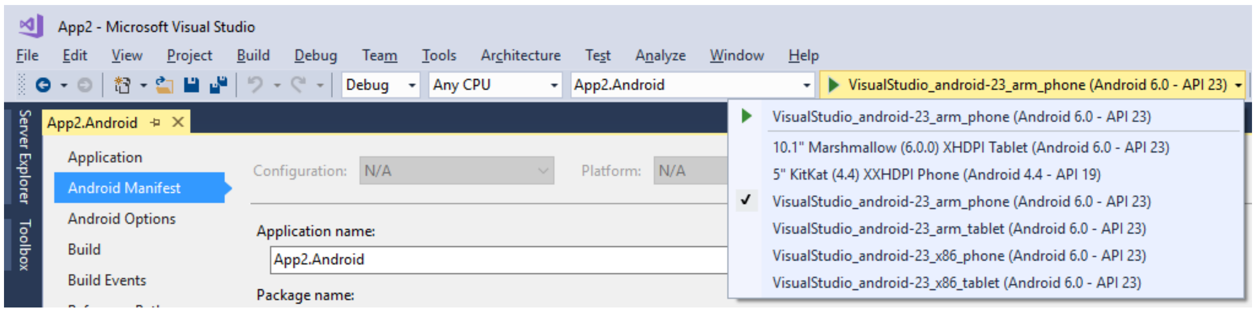
*Figure 14: Specifying capabilities in UWP projects*

The official documentation explains how to configure other options. However, remember that you need a paid subscription to the Windows Store in order to fill in the Packaging settings that you provide when preparing for publishing.

## Debugging and testing applications locally

Starting apps built with Xamarin.Forms for debugging and testing is easy. You simply select one of the platform projects in Solution Explorer as the startup project, then you select the target device and press **F5**. Do not forget to rebuild your solution before debugging for the first time. When you start an app for debugging, Visual Studio will build your solution and deploy the app package to the selected physical device or emulator. The result of the build process is an .apk file for Android, an .ipa file for iOS, and an .appx file for Windows 10.

When the app starts either on a physical device or on an emulator, Visual Studio attaches an instance of the debugger, and you will be able to use all the well-known, powerful debugging tools of the IDE, including (but not limited to) breakpoints, data tips, tool windows, watch windows, and more. The easiest way to select the target platform and configuration is by using the standard toolbar, which you can see in Figure 15 (notice that the list of emulators may vary, depending on your choices at installation time or on custom emulator images).


App2 - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Server Explorer Toolbox
App2.Android
Application
Android Manifest
Android Options
Build
Build Events

Debug | Any CPU | App2.Android | VisualStudio_android-23_arm_phone (Android 6.0 - API 23)

- VisualStudio_android-23_arm_phone (Android 6.0 - API 23)
- 10.1" Marshmallow (6.0.0) XHDPI Tablet (Android 6.0 - API 23)
- 5" KitKat (4.4) XXHDPI Phone (Android 4.4 - API 19)
- ✓ VisualStudio_android-23_arm_phone (Android 6.0 - API 23)
- VisualStudio_android-23_arm_tablet (Android 6.0 - API 23)
- VisualStudio_android-23_x86_phone (Android 6.0 - API 23)
- VisualStudio_android-23_x86_tablet (Android 6.0 - API 23)

Configuration: N/A  Platform: N/A

Application name: App2.Android

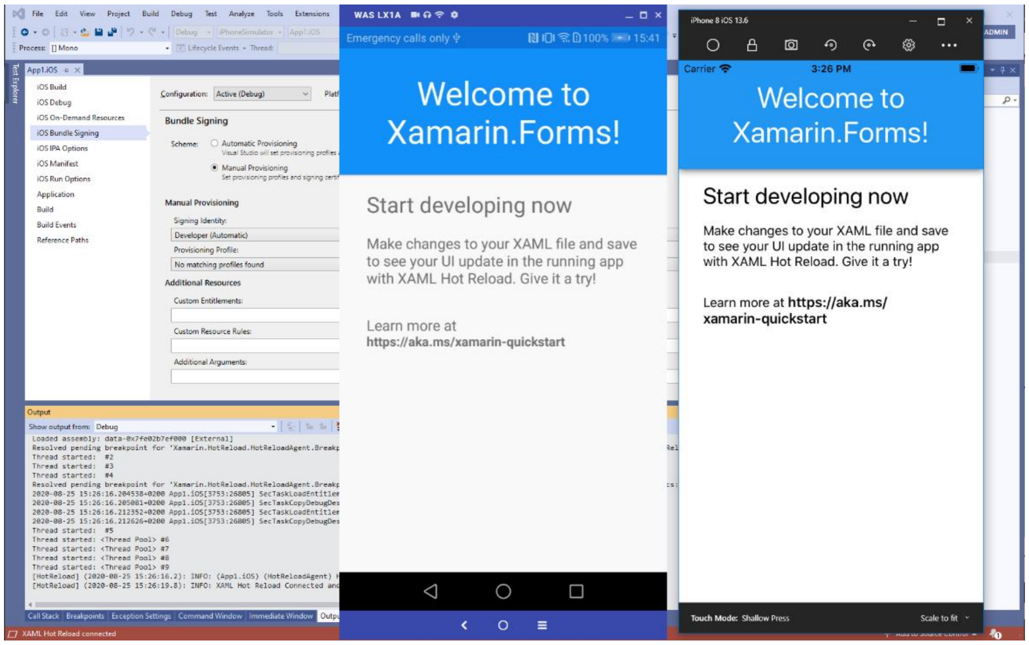
Package name:

*Figure 15: Selecting target platform and devices for debugging*

The **Debug** configuration is the appropriate choice during development. You will select **Ad-Hoc** when preparing for distribution on iOS and Android, or **Release** for Windows 10. The target architecture is normally **Any CPU for Android and Windows**, whereas it is **iPhoneSimulator** for debugging an iOS app in the iOS Simulator, or **iPhone** for debugging an app on a physical iPhone or iPad. (Remember that a physical Apple device must be associated to your Mac via Xcode and connected to the Mac, not the PC.) You can also quickly select the startup project and specify the target device. For example, in Figure 13 you can see a list of Android emulator configurations.

***Note: In this book, I will provide figures that show all the supported platforms in action when it is relevant to do so. In other cases, I will show just one platform in action, meaning that the same behavior is expected on all platforms.***

Figure 16 shows the previously created blank app running on iOS and Android platforms within the respective emulators. Notice how in the background, Visual Studio shows the **Output** window where you receive messages from the debugger; you will be able to use all the other debugging tools similarly.



Configuration: Active (Debug)

Bundle Signing
Scheme: Automatic Provisioning / Manual Provisioning

Manual Provisioning
Signing Identity: Developer (Automatic)
Provisioning Profile: No matching profiles found

Additional Resources
Custom Entitlements:
Custom Resource Rules:
Additional Arguments:

Output
Show output from: Debug

```
Loaded assembly: data-0x7fe02b7ef000 [External]
Resolved pending breakpoint for 'Xamarin.HotReload.HotReloadAgent.Breakp
Thread started:  #2
Thread started:  #3
Thread started:  #4
Resolved pending breakpoint for 'Xamarin.HotReload.HotReloadAgent.Breakp
2020-08-25 15:26:16.204538+0200 App1.iOS[3753:26805] SecTaskLoadEntitlem
2020-08-25 15:26:16.205081+0200 App1.iOS[3753:26805] SecTaskCopyDebugDes
2020-08-25 15:26:16.212352+0200 App1.iOS[3753:26805] SecTaskLoadEntitlem
2020-08-25 15:26:16.212626+0200 App1.iOS[3753:26805] SecTaskCopyDebugDes
Thread started:  #5
Thread started: <Thread Pool> #6
Thread started: <Thread Pool> #7
Thread started: <Thread Pool> #8
Thread started: <Thread Pool> #9
[HotReload] (2020-08-25 15:26:16.2): INFO: (App1.iOS) (HotReloadAgent) H
[HotReload] (2020-08-25 15:26:19.8): INFO: XAML Hot Reload Connected and
```

Welcome to Xamarin.Forms!

Start developing now

Make changes to your XAML file and save to see your UI update in the running app with XAML Hot Reload. Give it a try!

Learn more at https://aka.ms/xamarin-quickstart

*Figure 16: An app built with Xamarin.Forms running on Android and iOS*

As you can imagine, in order to build the iOS app, Visual Studio connected to the Mac and launched the Apple SDKs. In Figure 16, you see an example of the iOS Simulator, which deserves some more consideration.

### Setting up the iOS Simulator

Unlike Android and Windows, with which emulators run locally on your Windows development machine, the iOS Simulator runs on your Mac. However, if you have Visual Studio 2019, you can also download and install the Remoted iOS Simulator for Windows. When this is installed, the simulator will run on your Windows machine instead of running on the Mac.

The iOS Simulator on Windows is not enabled by default, so you need to open **Tools** > **Options** > **Xamarin** > **iOS Settings** and select the **Remote Simulator to Windows** option. An example of the iOS Simulator is available in Figure 14, but you will see others in the next chapters.

### Running apps on physical devices

Visual Studio can easily deploy an app package to physical Windows and Android devices connected to a Windows PC. For Android, you first need to enable the developer mode, which you accomplish with the following steps:

1. Open the **Settings** app.
2. Select the **About** item.
3. In the list that appears, locate the OS build number and tap this item seven times.

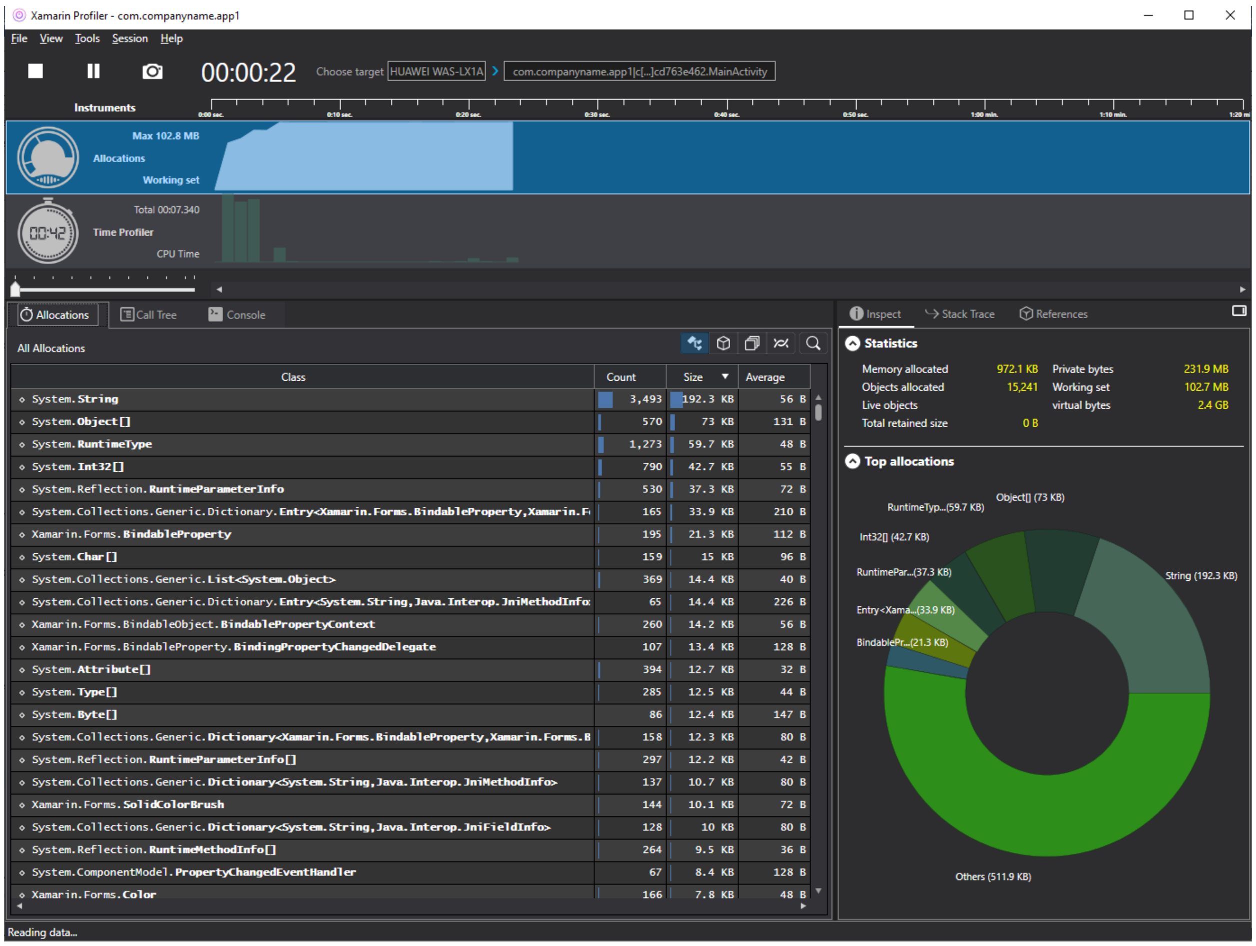
At this point, you can simply plug your device into the USB port of your PC, and Visual Studio will immediately recognize it. It will be visible in the list of available devices that you can see in Figure 15. For Windows, you first need to enable both your machine and your devices for development, and the official documentation provides guidance about this. Then you will be able to plug your devices into the USB port of your PC, and Visual Studio will recognize them as available target devices. For Apple mobile devices, you need to connect your iPhone or iPad to your Mac computer, making sure you make them discoverable through Xcode. Then, when you start debugging from Visual Studio, your app will be deployed to the iPhone or iPad through the Mac.

## Analyzing and profiling applications

The Xamarin toolbox has been recently enriched with two amazing tools: Xamarin Inspector and Xamarin Profiler. The Xamarin Inspector allows you to nicely inspect the visual tree of your Xamarin apps and make changes to the UI in real time (see the documentation).

The Xamarin Profiler is a complete suite of analysis tools in one program that you can use to profile your mobile apps and analyze performance, memory usage, CPU consumption, and more.

In Visual Studio, you can select **Tools** > **Xamarin Profiler**, and launch any platform version of your app for profiling, instead of pressing F5. After the app has been deployed to the selected device, and before starting up, the Profiler will ask you what kind of analysis you want to execute against the application. Figure 17 shows an example based on the selection of all the available instrumenting tools.



| Class | Count | Size | Average |
|---|---|---|---|
| System.String | 3,493 | 192.3 KB | 56 B |
| System.Object[] | 570 | 73 KB | 131 B |
| System.RuntimeType | 1,273 | 59.7 KB | 48 B |
| System.Int32[] | 790 | 42.7 KB | 55 B |
| System.Reflection.RuntimeParameterInfo | 530 | 37.3 KB | 72 B |
| System.Collections.Generic.Dictionary.Entry<Xamarin.Forms.BindableProperty,Xamarin.F | 165 | 33.9 KB | 210 B |
| Xamarin.Forms.BindableProperty | 195 | 21.3 KB | 112 B |
| System.Char[] | 159 | 15 KB | 96 B |
| System.Collections.Generic.List<System.Object> | 369 | 14.4 KB | 40 B |
| System.Collections.Generic.Dictionary.Entry<System.String,Java.Interop.JniMethodInfo | 65 | 14.4 KB | 226 B |
| Xamarin.Forms.BindableObject.BindablePropertyContext | 260 | 14.2 KB | 56 B |
| Xamarin.Forms.BindableProperty.BindingPropertyChangedDelegate | 107 | 13.4 KB | 128 B |
| System.Attribute[] | 394 | 12.7 KB | 32 B |
| System.Type[] | 285 | 12.5 KB | 44 B |
| System.Byte[] | 86 | 12.4 KB | 147 B |
| System.Collections.Generic.Dictionary<Xamarin.Forms.BindableProperty,Xamarin.Forms.B | 158 | 12.3 KB | 80 B |
| System.Reflection.RuntimeParameterInfo[] | 297 | 12.2 KB | 42 B |
| System.Collections.Generic.Dictionary<System.String,Java.Interop.JniMethodInfo> | 137 | 10.7 KB | 80 B |
| Xamarin.Forms.SolidColorBrush | 144 | 10.1 KB | 72 B |
| System.Collections.Generic.Dictionary<System.String,Java.Interop.JniFieldInfo> | 128 | 10 KB | 80 B |
| System.Reflection.RuntimeMethodInfo[] | 264 | 9.5 KB | 36 B |
| System.ComponentModel.PropertyChangedEventHandler | 67 | 8.4 KB | 128 B |
| Xamarin.Forms.Color | 166 | 7.8 KB | 48 B |

| | | | |
|---|---|---|---|
| Memory allocated | 972.1 KB | Private bytes | 231.9 MB |
| Objects allocated | 15,241 | Working set | 102.7 MB |
| Live objects | | virtual bytes | 2.4 GB |
| Total retained size | 0 B | | |

*Figure 17: Analyzing app performance with Xamarin Profiler*

You can also take and compare snapshots of the memory in different moments to see if memory allocation can cause potential problems. This is an excellent performance analysis tool, and the documentation will provide all the details you need to improve your app performance.

## Chapter summary

This chapter introduced the Xamarin.Forms platform and its goals, describing the required development tools and offering an overview of a Xamarin.Forms solution, passing through the platform projects and their most important settings. You have also seen how to start an app for debugging using emulators, and how to profile an app to improve performance. Now that you have an overview of Xamarin.Forms and the development tools, the next step is understanding what is at its core: sharing code across platforms.

# Chapter 2 Sharing Code Among Platforms

Xamarin.Forms allows you to build apps that run on Android, iOS, and Windows from a single C# codebase. This is possible because, at its core, Xamarin.Forms allows the sharing among platforms of all the code for the user interface and all the code that is not platform-specific. Starting with Visual Studio 2019, code sharing happens only through .NET Standard. This chapter explains how code sharing works in Xamarin.Forms, also providing suggestions on how you can adopt .NET Standard to share your own code and not only the user interface.

## Sharing code with .NET Standard

The .NET Standard provides a set of formal specifications for APIs that all the .NET development platforms, such as .NET Framework, .NET Core, and Mono, must implement. This allows for unifying .NET platforms and avoiding future fragmentation. By creating a .NET Standard library, you will ensure your code will run on any .NET platform without the need to select any targets. This also solves a common problem with portable libraries, since every portable library can target a different set of platforms, which implies potential incompatibility between libraries and projects. Microsoft has an interesting blog post about the .NET Standard and its goals and implementations that will clarify any doubts about this specification.

At the time of writing, version 2.1 of .NET Standard is available, though new Xamarin.Forms projects use version 2.0 by default. .NET Standard libraries are certainly not exclusive to Xamarin. In fact, they can be used in many other development scenarios. For example, a .NET Standard library could be used to share a Model-View-ViewModel architecture between a WPF project and a UWP project.

The most important characteristics of .NET Standard libraries are:

- They produce a compiled, reusable .dll assembly.
- They can reference other libraries and have dependencies such as NuGet packages.
- They can contain XAML files for the user interface definition and C# files.
- They cannot expose code that leverages specific APIs that are not available on all the platforms targeted by a specific .NET Standard version.
- They are a better choice when you need to implement architectural patterns such as MVVM, factory, inversion of control (IoC) with dependency injection, and service locator.
- With regard to Xamarin.Forms, they can use the service locator pattern to implement an abstraction and to invoke platform-specific APIs through platform projects (this will be discussed in Chapter 9).
- They are easier to unit test.

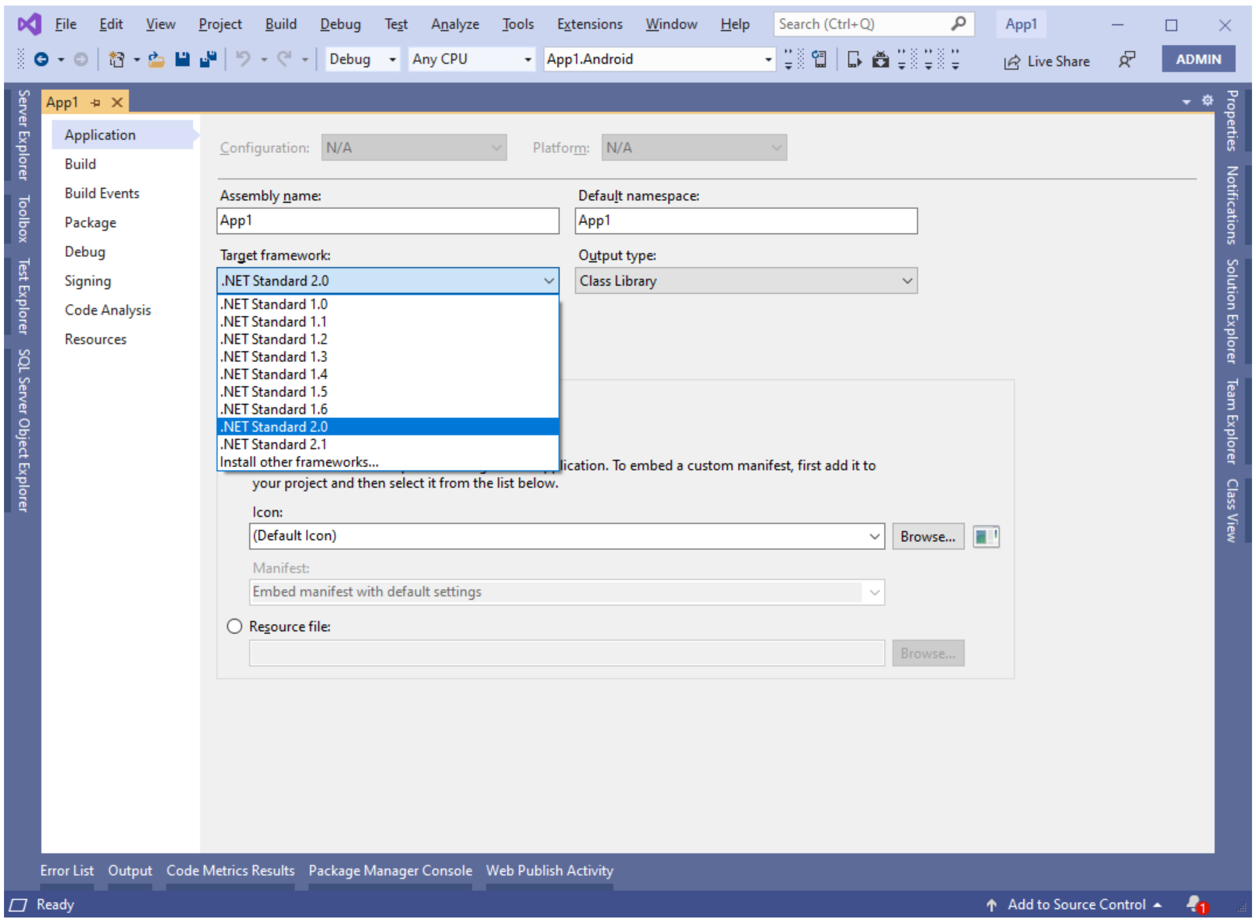
For example, it's only possible to use a component of the .NET Standard library for both a WPF and UWP application to access the location sensor of a device if the component is supported by both application types. Normally, you would create a .NET Standard library project manually, and then add the necessary references to and from other projects in the solution. In the case of Xamarin.Forms, Visual Studio 2019 automatically generates a .NET Standard project that is referenced by the platform projects in the solution, and that has a dependency on the Xamarin.Forms NuGet package.

## .NET Standard with Xamarin.Forms

In Xamarin.Forms solutions, the .NET Standard library contains all the cross-platform code and all files that are required to support the shared code. For example, it contains the user interface definition, the imperative code that handles and reacts to the user interface, images that must be embedded in the app package, resource files, and configuration files. As an implication, platform-specific code will not be included in the .NET Standard library; instead, it will be placed in the Android, iOS, and UWP projects.

For example, the way an app accesses a local SQLite database is platform-specific, so the code for this will not be in the .NET Standard library. Chapter 9 discusses platform-specific code in more detail. For now, it is important to focus on the .NET Standard project, because in the next chapters you will write all the code inside this project, and it will run on Android, iOS, and UWP from this single codebase.

By default, Xamarin.Forms solutions target .NET Standard 2.0. In case you need to address compatibility issues with third-party libraries, you can change the .NET Standard version in the project properties. To accomplish this, right-click the project name in Solution Explorer, and then select **Properties**. As you can see in Figure 18, you can change the version of .NET Standard in the **Target framework** drop-down menu.


Assembly name: App1
Default namespace: App1
Target framework: .NET Standard 2.0
Output type: Class Library

.NET Standard 1.0
.NET Standard 1.1
.NET Standard 1.2
.NET Standard 1.3
.NET Standard 1.4
.NET Standard 1.5
.NET Standard 1.6
.NET Standard 2.0
.NET Standard 2.1
Install other frameworks...

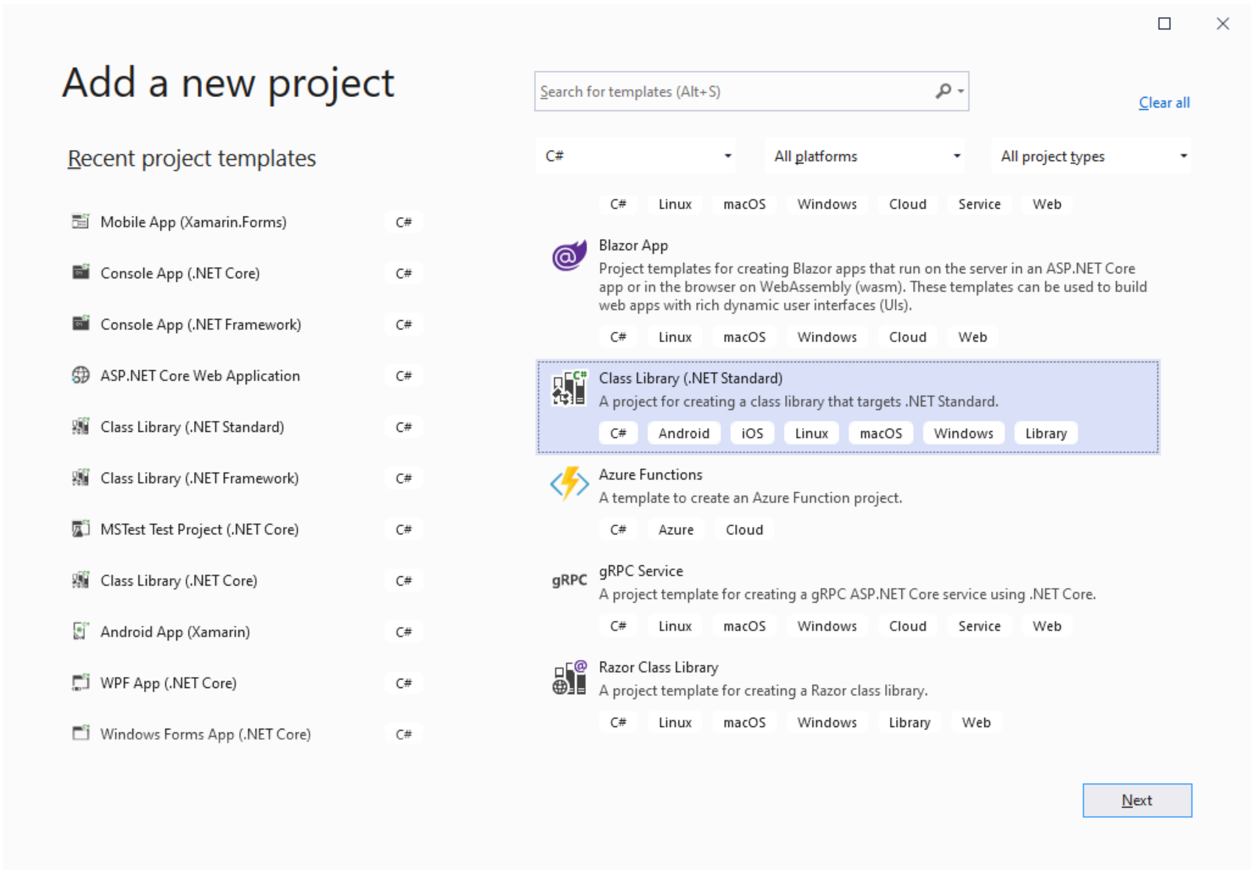
*Figure 18: Changing .NET Standard version*

When you change the version, Visual Studio will update the project accordingly and will reload the solution.

## Adding .NET Standard libraries

.NET Standard libraries make it easier to reuse code across platforms and solutions. You could create a reusable library for multiple solutions, for example, if you are implementing a service library to call Web APIs, or simply add a .NET Standard library to one solution to share more code across platforms.

To add a .NET Standard library to a Xamarin.Forms solution, right-click the solution name and then select **Add** > **New Project**. In the **Add a new project** dialog, you can filter the list of templates by language by selecting **C#** from the drop-down menu (see Figure 19).



Add a new project

Recent project templates

| | |
|---|---|
| Mobile App (Xamarin.Forms) | C# |
| Console App (.NET Core) | C# |
| Console App (.NET Framework) | C# |
| ASP.NET Core Web Application | C# |
| Class Library (.NET Standard) | C# |
| Class Library (.NET Framework) | C# |
| MSTest Test Project (.NET Core) | C# |
| Class Library (.NET Core) | C# |
| Android App (Xamarin) | C# |
| WPF App (.NET Core) | C# |
| Windows Forms App (.NET Core) | C# |

Search for templates (Alt+S)

Clear all

C# | All platforms | All project types

C# Linux macOS Windows Cloud Service Web

Blazor App
Project templates for creating Blazor apps that run on the server in an ASP.NET Core app or in the browser on WebAssembly (wasm). These templates can be used to build web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Cloud Web

Class Library (.NET Standard)
A project for creating a class library that targets .NET Standard.
C# Android iOS Linux macOS Windows Library

Azure Functions
A template to create an Azure Function project.
C# Azure Cloud

gRPC Service
A project template for creating a gRPC ASP.NET Core service using .NET Core.
C# Linux macOS Windows Cloud Service Web

Razor Class Library
A project template for creating a Razor class library.
C# Linux macOS Windows Library Web

Next

*Figure 19: Adding .NET Standard libraries to a Xamarin.Forms solution*

Locate and select the **Class Library (.NET Standard)** project template (see Figure 19). Simply click **Next**, provide a name for the new project, and click **Create**. After a few seconds, the new project will be added to the solution.

You will need to add a reference to this new project from each of the projects in the solution that will use its types. Discussing all the APIs provided by .NET Standard is not in the scope of this book, but the official documentation provides the proper information. For example, you will be able to click on the version number of each .NET Standard implementation to get a list of exposed APIs.

## Chapter summary

This chapter discussed code-sharing in Xamarin.Forms and explained how .NET Standard is leveraged to share user interface files and platform-independent code. .NET Standard libraries produce reusable assemblies, allow for implementing better architectures, and cannot contain platform-specific code.

.NET Standard libraries represent the present and future of code-sharing across platforms, are based on a formal set of API specifications, and they make sure your code will run on all the platforms that support the selected version of .NET Standard. Now that you have basic knowledge of how code sharing works and why it is fundamental, in the next chapters you will start writing code and building cross-platform user interfaces.

# Chapter 3 Building the User Interface with XAML

Xamarin.Forms is, at its core, a library that allows you to create native user interfaces from a single C# codebase by sharing code. This chapter provides the foundations for building the user interface in a Xamarin.Forms solution. Then, in the next three chapters, you will learn in more detail about layouts, controls, pages, and navigation.

## The structure of the user interface in Xamarin.Forms

The biggest benefit of Xamarin.Forms is that you can define the entire user interface of your application inside the .NET Standard project that you selected for sharing code. Native apps for iOS, Android, and Windows that you get when you build a solution will render the user interface with the proper native layouts and controls on each platform. This is possible because Xamarin.Forms maps native controls into C# classes that are then responsible for rendering the appropriate visual element, depending on the platform the app is running on. These classes actually represent visual elements such as pages, layouts, and controls.

Because the .NET Standard library can only contain code that will certainly run on all platforms, Xamarin.Forms maps only those visual elements common to all platforms. For instance, iOS, Android, and Windows all provide text boxes and labels; thus Xamarin.Forms can provide the **`Entry`** and **`Label`** controls that represent text boxes and labels, respectively. However, each platform renders and manages visual elements differently from one another, with different properties and behavior. This implies that controls in Xamarin.Forms expose only properties and events that are common to every platform, such as the font size and the text color.

In Chapter 9, you will learn how to use native controls directly, but for now let’s focus on how Xamarin.Forms allows the creation of user interfaces with visual elements provided out of the box. The user interface in iOS, Android, and Windows has a hierarchical structure made of pages that contain layouts that contain controls. Layouts can be considered as containers of controls that allow for dynamically arranging the user interface in different ways. Based on this consideration, Xamarin.Forms provides a number of page types, layouts, and controls that can be rendered on each platform.

When you create a Xamarin.Forms solution, the .NET Standard shared project will contain a root page that you can populate with visual elements. Then you can design a more complex user interface by adding other pages and visual elements. To accomplish this, you can use both C# and the Extensible Application Markup Language (XAML). Let’s discuss both methods. Furthermore, C# Markup, which is a new feature currently in preview, allows you to define the user interface in C# using new fluent APIs discussed in the last section of this chapter.

## Coding the user interface in C#

In Xamarin.Forms, you can create the user interface of an application in C# code. For instance, Code Listing 2 demonstrates how to create a page with a layout that arranges controls in a stack containing a label and a button. For now, do not focus on element names and their properties (they will be explained in the next chapter). Rather, focus on the hierarchy of visual elements that the code introduces.

*Code Listing 2*

```
var newPage = new ContentPage();
newPage.Title = "New page";

var newLayout = new StackLayout();
newLayout.Orientation = StackOrientation.Vertical;
newLayout.Padding = new Thickness(10);

var newLabel = new Label();
newLabel.Text = "Welcome to Xamarin.Forms!";

var newButton = new Button();
newButton.Text = "Tap here";
newButton.Margin = new Thickness(0, 10, 0, 0);

newLayout.Children.Add(newLabel);
newLayout.Children.Add(newButton);

newPage.Content = newLayout;
```

Here you have full IntelliSense support. However, as you can imagine, creating a complex user interface entirely in C# can be challenging for at least the following reasons:

- Representing a visual hierarchy made of tons of elements in C# code is extremely difficult.
- You must write the code in a way that allows you to distinguish between user interface definition and other imperative code.
- As a consequence, your C# becomes much more complex and difficult to maintain.

In the early days of Xamarin.Forms, defining the user interface could only be done in C# code. Fortunately, you now have a much more versatile way of designing the user interface with XAML, as you'll learn in the next section. Obviously, there are still situations in which you might need to create visual elements in C#, for example, if you need to add new controls at runtime, although this is the only scenario for which I suggest you code visual elements in C#.

## The modern way: Designing the user interface with XAML

XAML is the acronym for *eXtensible Application Markup Language*. As its name implies, XAML is a markup language that you can use to write the user interface definition in a declarative fashion. XAML is not new in Xamarin.Forms, since it was first introduced more than 10 years ago with Windows Presentation Foundation, and it has always been available in platforms such as Silverlight, Windows Phone, and the Universal Windows Platform.

XAML derives from XML and, among others, it offers the following benefits:

- XAML makes it easy to represent structures of elements in a hierarchical way, where pages, layouts, and controls are represented with XML elements and properties with XML attributes.
- It provides clean separation between the user interface definition and the C# logic.
- Being a declarative language separated from the logic, it allows professional designers to work on the user interface without interfering with the imperative code.

The way you define the user interface with XAML is unified across platforms, meaning that you design the user interface once and it will run on iOS, Android, and Windows.

***Note: XAML in Xamarin.Forms adheres to Microsoft's XAML 2009 specifications, but its vocabulary is different from XAML in other platforms, such as WPF or UWP. So, if you have experience with these platforms, you will notice many differences in how visual elements and their properties are named. Microsoft is working on unifying all .NET development platforms into a new framework called .NET MAUI, whose release is expected for November 2021, and it is reasonable to think that XAML vocabularies will also be unified. Also, remember that XAML is case sensitive for object names and their properties and members.***

For example, when you create a Xamarin.Forms solution, you can find a file in the .NET Standard project called **MainPage.xaml**, whose markup is represented in Code Listing 3.

*Code Listing 3*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">

    <StackLayout>
        <Frame BackgroundColor="#2196F3" Padding="24" CornerRadius="0">
            <Label Text="Welcome to Xamarin.Forms!"
             HorizontalTextAlignment="Center"
             TextColor="White" FontSize="36"/>
        </Frame>
        <Label Text="Start developing now" FontSize="Title"
         Padding="30,10,30,10"/>
        <Label Text="Make changes to your XAML file and save to see your
UI update in the running app with XAML Hot Reload. Give it a try!"
```

```
                FontSize="16" Padding="30,0,30,0"/>
        <Label FontSize="16" Padding="30,24,30,0">
            <Label.FormattedText>
                <FormattedString>
                    <FormattedString.Spans>
                        <Span Text="Learn more at "/>
                        <Span
                              Text="https://aka.ms/xamarin-quickstart"
                              FontAttributes="Bold"/>
                    </FormattedString.Spans>
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>

</ContentPage>
```

A XAML file in Xamarin.Forms normally contains a page or a custom view. The root element is a **ContentPage** object, which represents its C# class counterpart, and is rendered as an individual page. In XAML, the **Content** property of a page is implicit, meaning you do not need to write a **ContentPage.Content** element. The compiler assumes that the visual elements you enclose between the **ContentPage** tags are assigned to the **ContentPage.Content** property. The **StackLayout** is a container of views. The **Frame** view draws a border around a view with properties to control the distance (**Padding**) and the corner appearance (**CornerRadius**).

The **Label** element represents the **Label** class in C#. Properties of this class are assigned with XML attributes, such as **Text**, **VerticalOptions**, and **HorizontalOptions**. Properties can be assigned with complex types, and this must be done in a hierarchical way, not inline. An example in Code Listing 3 is in the assignment of the **Label.FormattedText property**, whose value is of type **FormattedString**, and whose attributes are part of the **Spans** object.

You probably already have the immediate perception of better organization and visual representation of the structure of the user interface. If you look at the root element, you can see a number of attributes whose definitions start with **xmlns**. These are referred to as XML namespaces and are important because they make it possible to declare visual elements defined inside specific namespaces or XML schemas. For example, **xmlns** points to the root XAML namespace defined inside a specific XML schema and allows for adding to the UI definition all the visual elements defined by Xamarin.Forms; **xmlns:x** points to an XML schema that exposes built-in types; and **xmlns:local** points to the app's assembly, making it possible to use objects defined in your project.

Each page or layout can only contain one visual element. In the case of the autogenerated MainPage.xaml page, you cannot add other visual elements to the page unless you organize them into a layout. This is the reason why in Code Listing 3 you see several visual elements inside a **StackLayout**.

Let's simplify the auto-generated code to have some text and a button. This requires us to have a **Button** below a **Label**, and both must be wrapped inside a container such as the **StackLayout**, as demonstrated in Code Listing 4.

***Tip: IntelliSense will help you add visual elements faster by showing element names and properties as you type. You can then simply press Tab or double-click to quickly insert an element.***

*Code Listing 4*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="10">
        <Label Text="Welcome to Xamarin Forms!"
       VerticalOptions="Center"
       HorizontalOptions="Center" />

        <Button x:Name="Button1" Text="Tap here!"
            Margin="0,10,0,0"/>
    </StackLayout>

</ContentPage>
```

If you did not include both controls inside the layout, Visual Studio will raise an error. You can nest other layouts inside a parent layout and create complex hierarchies of visual elements. Notice the **`x:Name`** assignment for the **`Button`**. Generally speaking, with **`x:Name`** you can assign an identifier to any visual element so that you can interact with it in C# code, for example, if you need to retrieve a property value.

If you have never seen XAML before, you might wonder how you can interact with visual elements in C# at this point. In Solution Explorer, if you expand the **MainPage.xaml** file, you will see a nested file called **MainPage.xaml.cs**. This is the code-behind file, and it contains all the imperative code for the current page. In this case, the simplest form of a code-behind file, the code contains the definition of the **`MainPage`** class, which inherits from **`ContentPage`**, and the page constructor, which makes an invocation to the **`InitializeComponent`** method of the base class and initializes the page.

You will access the code-behind file often from Solution Explorer, but Visual Studio 2019 offers another easy way that is related to a very common requirement: responding to events raised by the user interface.

## Productivity features for XAML IntelliSense

The XAML code editor in Visual Studio 2019 is powered by the same engine that is behind Windows Presentation Foundation (WPF) and Universal Windows Platform (UWP). This is extremely important for several reasons:

- **Full, rich IntelliSense support**: Earlier versions of IntelliSense could make writing XAML markup painful to do. The new version adds linting and fuzzy matching, described in more detail shortly.
- **Quick actions and refactorings**: Light bulb suggestions are now available to XAML in Xamarin.Forms, just as they have been for WPF and UWP, making it easy to resolve XML namespaces, remove unused namespaces, and organize code with contextualized suggestions.
- **Go To Definition and Peek Definition**: These popular features, previously available only to the C# code editor, are also now available to Xamarin.Forms XAML.
- **Enhanced support for binding expressions**: IntelliSense now lists available objects for bindings based on the {Binding} markup extension, and also lists available resources when using the {StaticResource} markup extension. Hints about this feature will be provided in Chapter 7, Resources and Data Binding.

I will now describe these productivity features in more detail.

## Fuzzy matching and linting

Fuzzy matching is a feature that helps you find appropriate completions based on what you type. For example, if you type **`Stk`** and then press the Tab key, IntelliSense will add a **`StackLayout`** tag. This feature is also capable of providing a list of possible completions as you type in a control name. For example, if you type **`Layout`**, IntelliSense will offer **`StackLayout`**, **`FlexLayout`**, **`AbsoluteLayout`**, and **`RelativeLayout`** as possible completions, as well as closing tags based on the same typing.

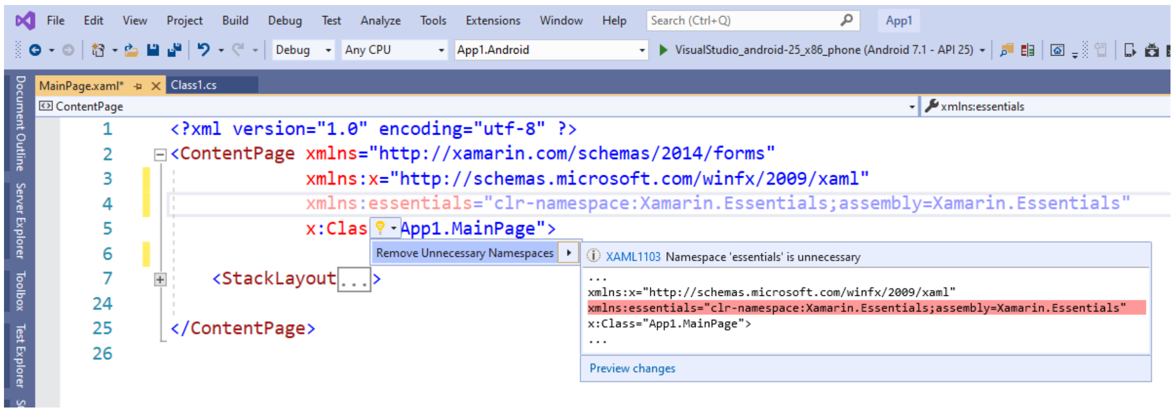
Another interesting feature of fuzzy matching is CamelCase matching, which provides shortcuts based on CamelCase types. For instance, if you type **`RL`** and then press Tab, the editor inserts a **`RelativeLayout`** tag. With linting, the code editor underlines code issues as you type with red squiggles (critical errors) or green squiggles (warnings).

## Light bulb: Quick actions and refactorings

The light bulb icon was first introduced to IntelliSense in Visual Studio 2015 for the C# and Visual Basic languages, and it was later added to XAML IntelliSense in WPF and UWP. Now this useful feature is also available to XAML in Xamarin.Forms. With this tool, when a code issue is detected, you can click the light bulb icon (or press Ctrl + .), and IntelliSense will show potential fixes for that code issue. In Xamarin.Forms XAML, the light bulb can suggest code fixes to import missing XML namespaces, sort XML namespaces, and remove unused XML namespaces.

***Tip: Unused XML namespaces have a lighter color in the code editor, so they are more easily recognizable. When you hover over unused XML namespaces, the light bulb will be available, and a quick action will suggest to remove all unused namespaces as a potential fix.***

Figure 20 provides an example based on a declared but never used XML namespace.



```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:essentials="clr-namespace:Xamarin.Essentials;assembly=Xamarin.Essentials"
             x:Clas    App1.MainPage">

    <StackLayout...>

</ContentPage>
```

XAML1103 Namespace 'essentials' is unnecessary

```
...
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
xmlns:essentials="clr-namespace:Xamarin.Essentials;assembly=Xamarin.Essentials"
x:Class="App1.MainPage">
...
```

*Figure 20: Quickly removing an unused XML namespace*

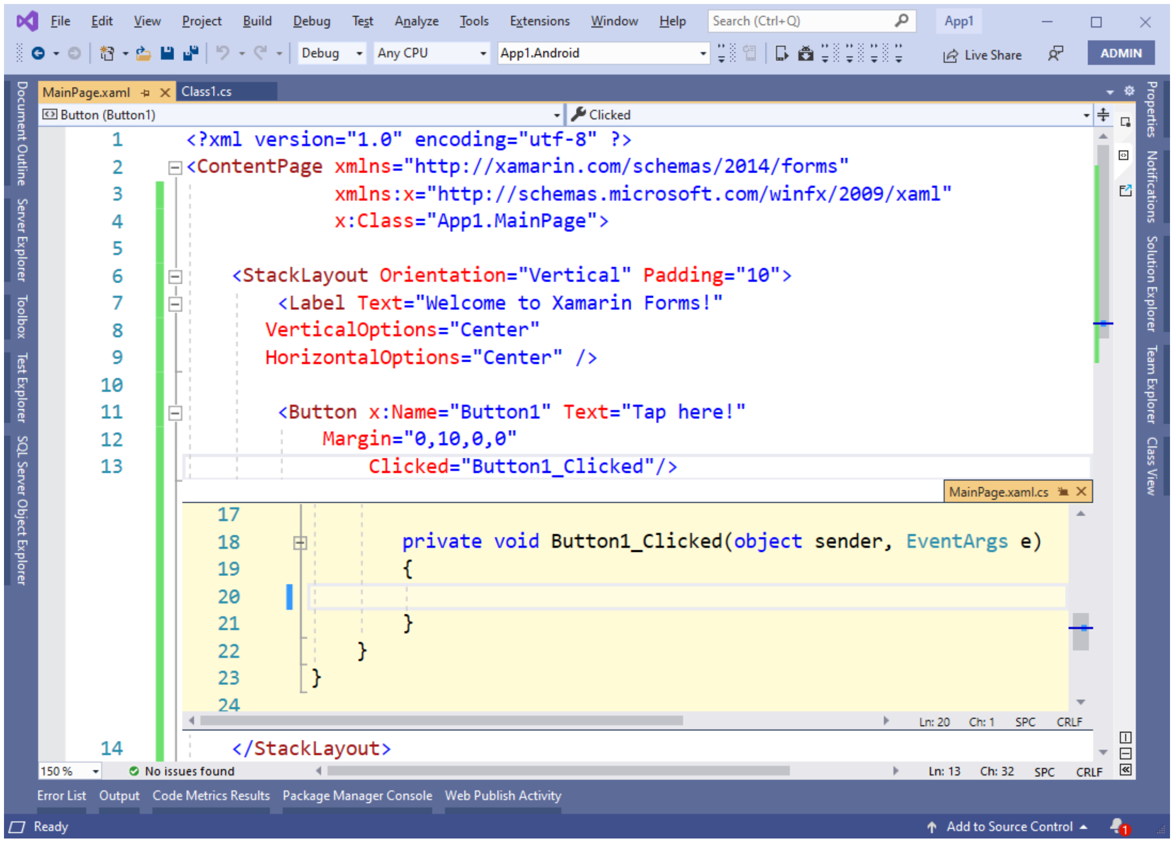
## Go To Definition and Peek Definition

Go To Definition and Peek Definition are popular, extremely useful features in the code editor, and they are also available in the XAML IntelliSense in Xamarin.Forms. Both are available through the context menu when you right-click an object name in the code editor.

With Go To Definition, Visual Studio will open the definition of the selected object and, if it is defined in a different file, such a file will be opened in a separate tab. The cursor will be moved to the object definition. In XAML, this is particularly useful when you need to go to the definition of objects such as styles, templates, and other resources that might be defined in a different file.

Peek Definition opens an interactive pop-up window in the active editor, allowing you to see the definition of an object or to make edits without leaving the active window. Additionally, you are not limited to viewing or editing objects defined in a XAML file, as you can also peek at the definition of an object defined in the C# code-behind file.

Figure 21 shows an example of Peek Definition where a C# event handler for the **`Clicked`** event of a **`Button`** is displayed within a XAML editor window.


```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="10">
        <Label Text="Welcome to Xamarin Forms!"
        VerticalOptions="Center"
        HorizontalOptions="Center" />

        <Button x:Name="Button1" Text="Tap here!"
            Margin="0,10,0,0"
                Clicked="Button1_Clicked"/>
```

```
        private void Button1_Clicked(object sender, EventArgs e)
        {

        }
    }
}
```

```
    </StackLayout>
```

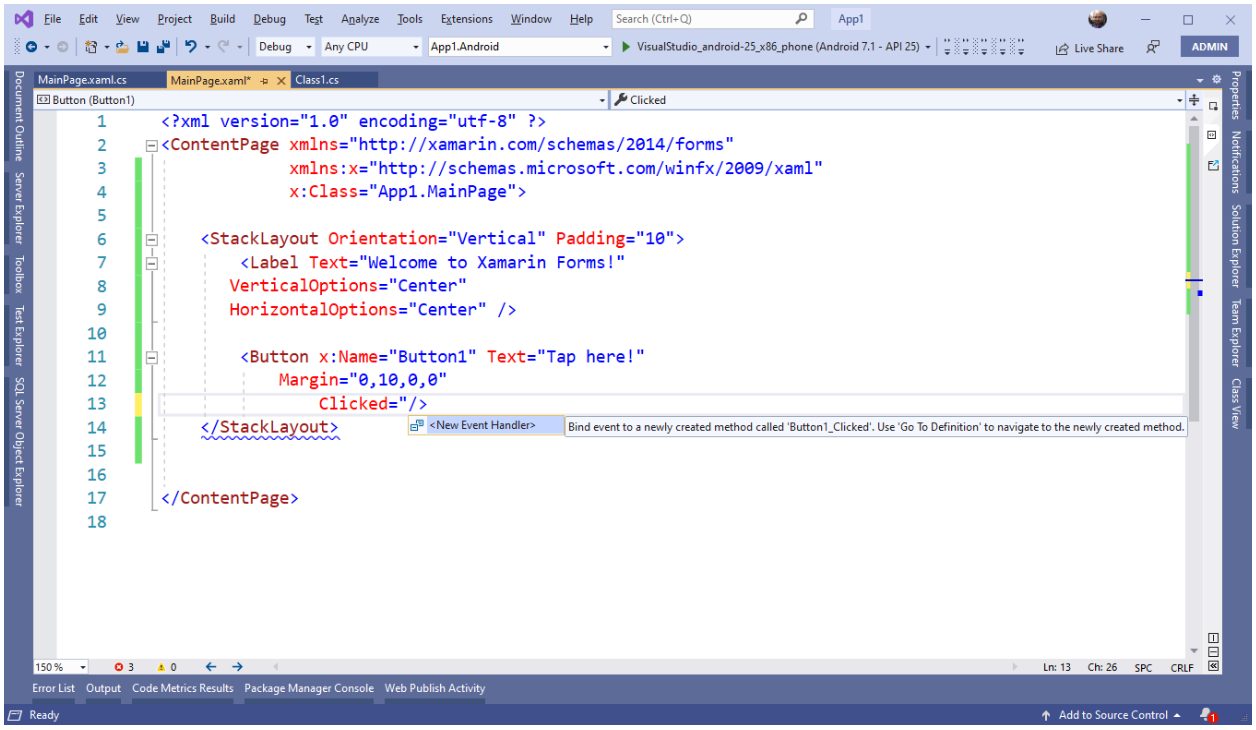
Figure 21: Making edits in the active editor with Peek Definition

## Responding to events

Events are fundamental for the interaction between the user and the application, and controls in Xamarin.Forms raise events as normally happens in any platform. Events are handled in the C# code-behind file. Visual Studio 2019 makes it simple to create event handlers with an evolved IntelliSense experience. For instance, suppose you want to perform an action when the user taps the button defined in the previous code. The **`Button`** control exposes an event called **`Clicked`** that you assign the name of an event handler as follows.

```
<Button x:Name="Button1" Text="Tap here!" Margin="0,10,0,0"
 Clicked="Button1_Clicked"/>
```

However, when you type **`Clicked="`**, Visual Studio offers a shortcut that allows the generation of an event handler in C# based on the control's name, as shown in Figure 22.



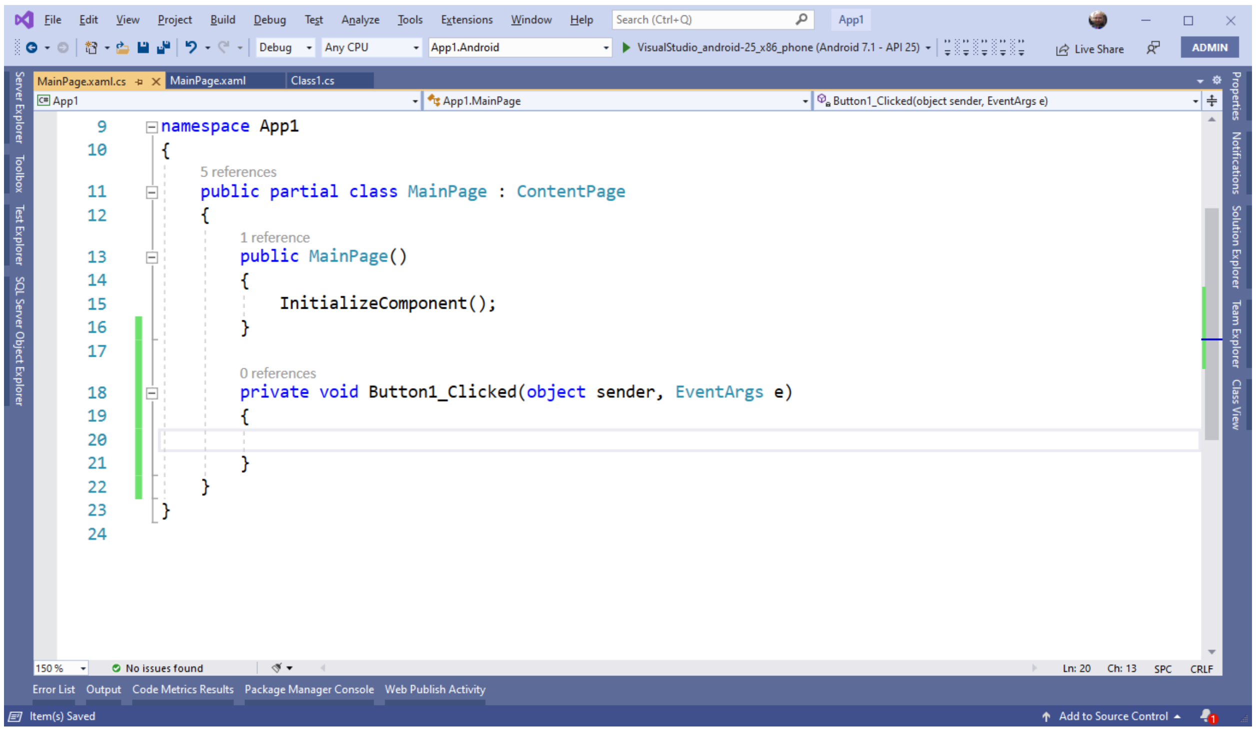
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="10">
        <Label Text="Welcome to Xamarin Forms!"
     VerticalOptions="Center"
     HorizontalOptions="Center" />

        <Button x:Name="Button1" Text="Tap here!"
            Margin="0,10,0,0"
                Clicked="/>
    </StackLayout>


</ContentPage>
```

<New Event Handler>

Bind event to a newly created method called 'Button1_Clicked'. Use 'Go To Definition' to navigate to the newly created method.

Figure 22: Generating an event handler

If you press **Tab**, Visual Studio will insert the name of the new event handler and generate the C# event handler in the code-behind. You can quickly go to the event handler by right-clicking its name and then selecting **Go To Definition**. You will be redirected to the event handler definition in the C# code-behind, as shown in Figure 23.



```
namespace App1
{
    5 references
    public partial class MainPage : ContentPage
    {
        1 reference
        public MainPage()
        {
            InitializeComponent();
        }

        0 references
        private void Button1_Clicked(object sender, EventArgs e)
        {

        }
    }
}
```

Figure 23: The event handler definition in C#

At this point, you will be able to write the code that performs the action you want to execute, exactly as it happens with other .NET platforms such as WPF or UWP. Generally speaking, event handlers' signatures require two parameters: one of type **`object`** representing the control that raised the event, and one object of type **`EventArgs`** containing information about the event.

In many cases, event handlers work with derived versions of **`EventArgs`**, but these will be highlighted when appropriate. As you can imagine, Xamarin.Forms exposes events that are commonly available on all the supported platforms.

## Understanding type converters

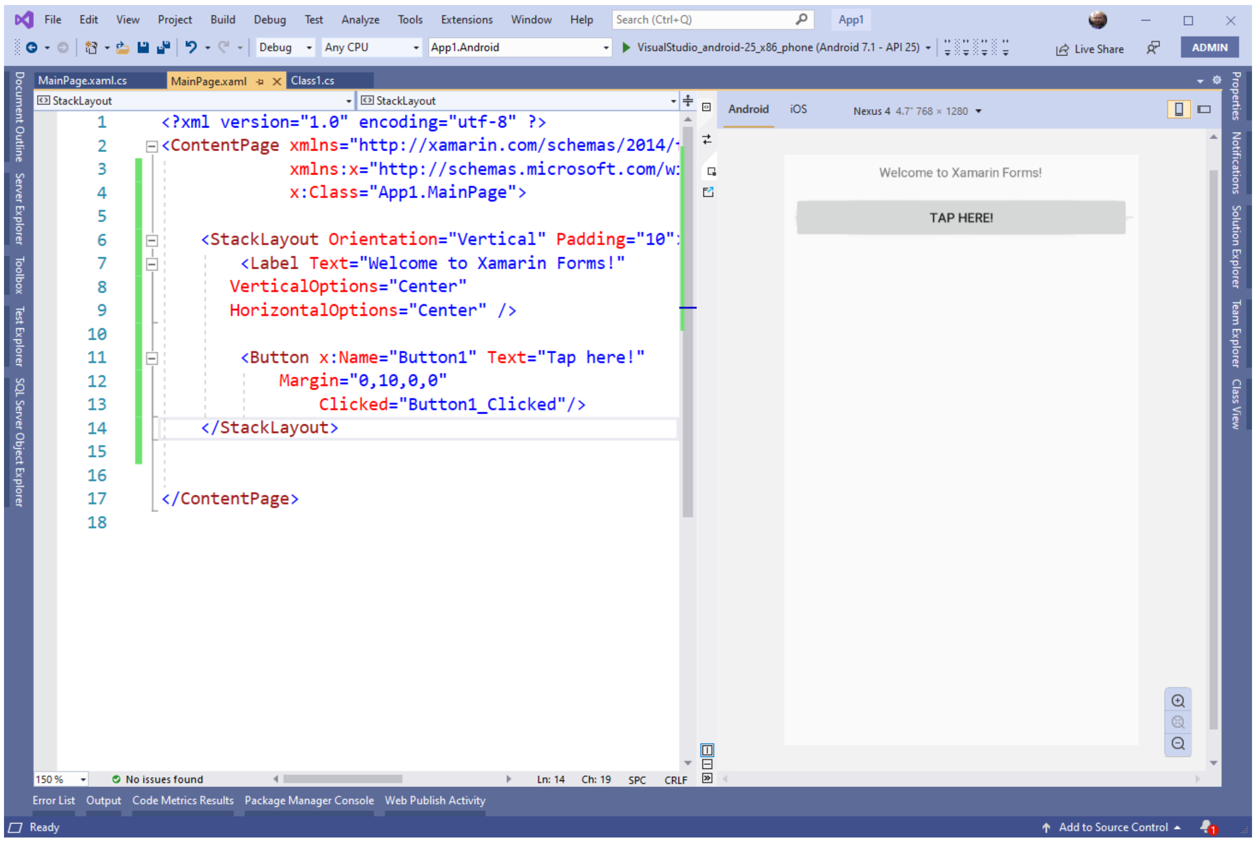
If you look at Code Listing 2, you will see that the **`Orientation`** property of the **`StackLayout`** is of type **`StackOrientation`**, the **`Padding`** property is of type **`Thickness`**, and the **`Margin`** property assigned to the **`Button`** is also of type **`Thickness`**. However, as you can see in Code Listing 4, the same properties are assigned with values passed in the form of strings in XAML.

Xamarin.Forms (and all the other XAML-based platforms) implement the so-called *type converters*, which automatically convert a string into the appropriate value for a number of known types. Summarizing all the available type converters and known target types is neither possible nor necessary at this point; you simply need to remember that, in most cases, strings you assign as property values are automatically converted into the appropriate type on your behalf.

## Xamarin.Forms Previewer

Xamarin.Forms doesn’t have a designer that allows you to draw the user interface visually with the mouse, the toolbox, and interactive windows as you are used to doing with platforms such as WPF, Windows Forms, and UWP. Though you can drag visual elements from the toolbox onto a XAML file, you still need to write all your XAML manually. However, Visual Studio 2019 offers the Xamarin.Forms Previewer, which shows a preview of the user interface in real time, as you edit your XAML.

Figure 24 shows the Xamarin.Forms Previewer in action. You enable the Xamarin.Forms Previewer by clicking either the **Vertical Split** or **Horizontal Split** buttons, which are located at the bottom-right corner of the code editor.



```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/
             xmlns:x="http://schemas.microsoft.com/w
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="10"
        <Label Text="Welcome to Xamarin Forms!"
       VerticalOptions="Center"
       HorizontalOptions="Center" />

        <Button x:Name="Button1" Text="Tap here!"
            Margin="0,10,0,0"
                Clicked="Button1_Clicked"/>
    </StackLayout>


</ContentPage>
```

File Edit View Project Build Debug Test Analyze Tools Extensions Window Help
Search (Ctrl+Q)
App1
Debug
Any CPU
App1.Android
VisualStudio_android-25_x86_phone (Android 7.1 - API 25)
Live Share
ADMIN
MainPage.xaml.cs
MainPage.xaml
Class1.cs
StackLayout
Android
iOS
Nexus 4 4.7" 768 × 1280
Welcome to Xamarin Forms!
TAP HERE!
150 %
No issues found
Ln: 14
Ch: 19
SPC
CRLF
Error List Output Code Metrics Results Package Manager Console Web Publish Activity
Ready
Add to Source Control

*Figure 24: The Xamarin.Forms Previewer*

***Tip: Remember to rebuild your solution before opening the Xamarin.Forms Previewer for the first time.***

At the bottom-right corner, the Previewer provides zoom controls. At the top, you can select the device factor (phone or tablet), the platform used to render the preview (Android or iOS), and the orientation (vertical or horizontal). If you wish to render the preview based on iOS, remember that you need Visual Studio to be connected to a Mac. If there are any errors in your XAML or if, for any reason, the Previewer is unable to render the preview, it will show a detailed error message.

The Xamarin.Forms Previewer is an important tool, because it prevents the need to run the application every time you make significant edits to the UI, as was required in the past. In the next chapters, I will often use the Previewer to demonstrate how the UI looks instead of running the emulators.

### Xamarin.Forms Hot Reload

In the past, if you needed to make one or more changes to the XAML code, you had to stop debugging, make your changes, and then restart the application. This was one of the biggest limitations in the Xamarin.Forms development experience. In Visual Studio 2019, this problem has been solved by a feature called Hot Reload.

The way it works is very simple: with the application running in debugging mode, you can make changes to your XAML, then you save your changes, and the application will redraw the user interface based on your changes. Hot Reload is available to all the supported development platforms. Hot Reload is enabled by default, but you can disable this feature by accessing **Tools > Options > Xamarin > Hot Reload** and disabling the **Enable XAML Hot Reload for Xamarin.Forms** option.

In the same dialog, you will find additional settings called **Reload Options**. Here you can change the way the user interface is redrawn when you make real-time changes to your XAML. In fact, by default, Hot Reload redraws the whole page, even if you change only one control property. It is possible to change this setting and make Hot Reload redraw only the view that was changed; however, this feature is still in preview. Hot Reload requires you to set the linker option for all the project to **Don't Link** or **Link None**. For iOS, you also need to enable the Mono interpreter in the project options.

Among the known limitations, Hot Reload might not work well when making changes to the visual hierarchy of the Shell (discussed in Chapter 6), and with views that are referenced by other views via the **`x:Name`** tag. Without a doubt, Hot Reload is a feature that will save you a lot of time, making it simple to understand the result of your changes at debugging time.

## Introducing C# Markup

XAML is the most efficient way to organize the user interface in Xamarin.Forms, but this is a declarative mode. Sometimes you might need to create views at runtime and add them to the visual tree when the application is running. Of course, this can be done in C#, following what you have learned previously in this chapter. However, the C# approach makes it more difficult to set some views' properties, such as data-binding assignments and the position inside the visual tree.

Starting with Xamarin.Forms 4.6, Microsoft introduced C# Markup, a new feature that allows for simplyfing the way the user interface can be defined in C# with fluent APIs. At the time of this writing, C# Markup is in preview and requires the following line of code inside the **`App`** constructor of the **App.xaml.cs** file, after the invocation to the **`InitializeComponent`** method:

```
Device.SetFlags(new string[]{ "Markup_Experimental" });
```

An example will help you better understand how this works. Some of the views used in this example will be discussed in more detail in Chapters 4 and 5, but the focus now is on the new Fluent APIs. Let's consider the following XAML code snippet that defines a **Grid** layout with a **Label** and an **Entry** inside. The **Grid**, discussed in the next chapter, is a layout that allows for organizing visual elements into rows and columns:

```
<ContentPage>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition />            <RowDefinition />
        </Grid.RowDefinitions>
        <Label Text="Phone number:"/>
        <Entry Keyboard="Telephone" Text="{Binding PhoneNumber}"
               Placeholder="Enter phone number" Grid.Row="1"/>
    </Grid>
</ContentPage>
```

This **Grid** has two rows: in the first row there is a **Label** that shows some text, and the second row includes an **Entry**, the equivalent of a text box, which allows entering a phone number, and therefore shows the appropriate keyboard and a placeholder text. The **Entry** is in the second row of the **Grid** (index start at 0), and its value is data-bound to a property **PhoneNumber** of a class that represents the data model. The **data** variable represents an instance of a data model. In C#, you would rewrite this UI as follows:

```
var grid1 = new Grid();
grid1.RowDefinitions.Add(new RowDefinition());
grid1.RowDefinitions.Add(new RowDefinition());
var phoneLabel = new Label { Text = "Phone number:", Margin = 5 };
var phoneEntry = new Entry { Keyboard = Keyboard.Telephone,
    Placeholder = "Enter phone number", Margin = 5 };
phoneEntry.SetBinding(Entry.TextProperty,
    new Binding(nameof(data.PhoneNumber)));
grid1.Children.Add(phoneLabel, 0, 0);
grid1.Children.Add(phoneEntry, 0, 1);
Content = grid1;
```

Since this piece of user interface is very simple, the code is also not complex, but you can see how you need to invoke several methods to add data binding and to place the controls inside the appropriate place. With C# Markup, you could rewrite this UI as follows:

```
Content = new Grid
{
    RowDefinitions =
    {
        new RowDefinition(),
        new RowDefinition()
    },
    Children =
    {
        new Label { Text = "Phone number:" }.Row(0),
        new Entry { Keyboard=Keyboard.Telephone,
            Placeholder="Enter phone number"}.
            Row(1).Bind(nameof(data.PhoneNumber))
    }
};
```

As you can see, not only is the visual representation of the code clearer, but here the **`Row`** extension method allows for quickly placing views inside the appropriate grid rows, and the **`Bind`** method sets data binding directly inline. If you consider that the number of available extension methods is huge, and that these target several Xamarin.Forms types, including styles, converters, gesture recognizers, and layouts, it is easy to understand how this will simplify the addition of visual elements at runtime. Because this feature is still in preview, I recommend you have a look at the official documentation for information about the release date and about additional scenarios that might change over time.

## Chapter summary

Sharing the user interface across platforms is the main goal of Xamarin.Forms, and this chapter provided a high-level overview of how you define the user interface with XAML, based on a hierarchy of visual elements.

You have seen how to add visual elements and how to assign their properties; how type converters allow for passing string values in XAML, and how the compiler converts them into the appropriate types. You had a first look at the Xamarin.Forms Previewer to get a real-time, integrated representation of the user interface as you edit your XAML.

Finally, you had a quick look at an upcoming feature, C# Markup, which allows coding the user interface in managed code with Fluent APIs. Following this overview of how the user interface is defined in Xamarin.Forms, we will discuss important UI concepts in more detail, and we will start by organizing the user interface with layouts.

# Chapter 4 Organizing the UI with Layouts

Mobile devices such as phones, tablets, and laptops have different screen sizes and form factors. They also support both landscape and portrait orientations. Therefore, the user interface in mobile apps must dynamically adapt to the system, screen, and device so that visual elements can be automatically resized or rearranged based on the form factor and device orientation. In Xamarin.Forms, this is accomplished with layouts, which is the topic of this chapter.

## Understanding the concept of layout

***Tip: If you have previous experience with WPF or UWP, the concept of layout is the same as the concept of panels such as the `Grid` and the `StackPanel`.***

One of the goals of Xamarin.Forms is to provide the ability to create dynamic interfaces that can be rearranged according to the user's preferences or to the device and screen size. Because of this, controls in mobile apps you build with Xamarin should not have a fixed size or position on the UI, except in a very limited number of scenarios. To make this possible, Xamarin.Forms controls are arranged within special containers, known as *layouts*. Layouts are classes that allow for arranging visual elements in the UI, and Xamarin.Forms provides many of them.

In this chapter, you'll learn about available layouts and how to use them to arrange controls. The most important thing to keep in mind is that controls in Xamarin.Forms have a hierarchical logic; therefore, you can nest multiple panels to create complex user experiences. Table 1 summarizes the available layouts. You'll learn about them in more detail in the sections that follow.

*Table 1: Layouts in Xamarin.Forms*

| Layout | Description |
|---|---|
| **`StackLayout`** | Allows you to place visual elements near each other horizontally or vertically. |
| **`FlexLayout`** | Allows you to place visual elements near each other horizontally or vertically. Wraps visual elements to the next row or column if not enough space is available. |
| **`Grid`** | Allows you to organize visual elements within rows and columns. |
| **`AbsoluteLayout`** | A layout placed at a specified, fixed position. |

| Layout | Description |
|---|---|
| **`RelativeLayout`** | A layout whose position depends on relative constraints. |
| **`ScrollView`** | Allows you to scroll the visual elements it contains. |
| **`Frame`** | Draws a border and adds space around the visual element it contains. |
| **`ContentView`** | A special layout that can contain hierarchies of visual elements and can be used to create custom controls in XAML. |
| **`ControlTemplate`** | A group of styles and effects for controls, which is also the base for built-in views and custom views. |
| **`TemplatedView`** | The base class for **`ContentView`** objects that displays content within a control template. |
| **`ContentPresenter`** | Specifies where content appears inside a control template. |

Remember that only one root layout is assigned to the **`Content`** property of a page, and that layout can then contain nested visual elements and layouts. **`ControlTemplate`**, **`TemplatedView`**, and **`ContentPresenter`** are actually of more interest when you create your own custom views and will not be covered in this chapter. More information can be found in the documentation page.

## Alignment and spacing options

As a general rule, both layouts and controls can be aligned by assigning the **`HorizontalOptions`** and **`VerticalOptions`** properties with one of the property values from the **`LayoutOptions`** structure, summarized in Table 2. Providing an alignment option is very common. For instance, if you only have the root layout in a page, you will want to assign **`VerticalOptions`** with **`StartAndExpand`** so that the layout gets all the available space in the page. (Remember this consideration when you experiment with layouts and views in this chapter and the next one.)

*Table 2: Alignment options in Xamarin.Forms*

| Alignment | Description |
|---|---|
| **`Center`** | Aligns the visual element at the center. |
| **`CenterAndExpand`** | Aligns the visual element at the center and expands its bounds to fill the available space. |
| **`Start`** | Aligns the visual element at the left. |
| **`StartAndExpand`** | Aligns the visual element at the left and expands its bounds to fill the available space. |
| **`End`** | Aligns the visual element at the right. |
| **`EndAndExpand`** | Aligns the visual element at the right and expands its bounds to fill the available space. |
| **`Fill`** | Makes the visual element have no padding around itself and it does not expand. |
| **`FillAndExpand`** | Makes the visual element have no padding around itself and it expands to fill the available space. |

You can also control the space between visual elements with three properties: **`Padding`**, **`Spacing`**, and **`Margin`**, summarized in Table 3.

*Table 3: Spacing options in Xamarin.Forms*

| Spacing | Description |
|---|---|
| **`Margin`** | Represents the distance between the current visual element and its adjacent elements with either a fixed value for all four sides, or with comma-separated values for the left, top, right, and bottom. It is of type **`Thickness`**, and XAML has built in a type converter for it. |
| **`Padding`** | Represents the distance between a visual element and its child elements. It can be set with either a fixed value for all four sides, or with comma-separated values for the left, top, right, and bottom. It is of type **`Thickness`**, and XAML has built in a type converter for it. |

| Spacing | Description |
|---|---|
| **`Spacing`** | Available only in the **`StackLayout`** container, it allows you to set the amount of space between each child element, with a default of 6.0. |

I recommend you spend some time experimenting with how alignment and spacing options work in order to understand how to get the desired result in your user interfaces.

## The StackLayout

The **`StackLayout`** container allows the placing of controls near each other, as in a stack that can be arranged both horizontally and vertically. As with other containers, the **`StackLayout`** can contain nested panels. The following code shows how you can arrange controls horizontally and vertically. Code Listing 5 shows an example with a root **`StackLayout`** and two nested layouts.

*Code Listing 5*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical">
        <StackLayout Orientation="Horizontal" Margin="5">
            <Label Text="Sample controls" Margin="5"/>
            <Button Text="Test button" Margin="5"/>
        </StackLayout>
        <StackLayout Orientation="Vertical" Margin="5">
            <Label Text="Sample controls" Margin="5"/>
            <Button Text="Test button" Margin="5"/>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

The result of the XAML in Code Listing 5 is shown in Figure 25.

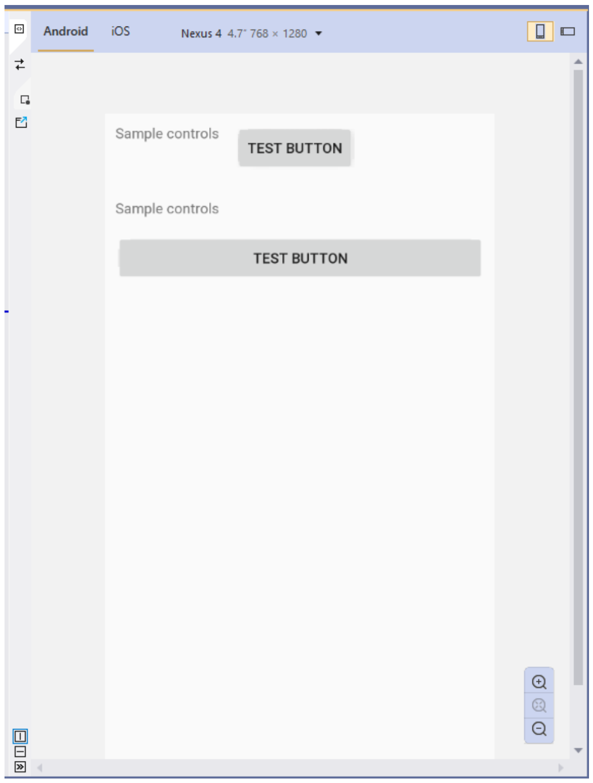


*Figure 25: Arranging visual elements with the StackLayout*

The **`Orientation`** property can be set as **`Horizontal`** or **`Vertical`**, and this influences the final layout. If not specified, **`Vertical`** is the default. One of the main benefits of XAML code is that element names and properties are self-explanatory, and this is the case in **`StackLayout`**'s properties, too.

Remember that controls within a **`StackLayout`** are automatically resized according to the orientation. If you do not like this behavior, you need to specify **`WidthRequest`** and **`HeightRequest`** properties on each control, which represent the width and height, respectively. **`Spacing`** is a property that you can use to adjust the amount of space between child elements; this is preferred to adjusting the space on the individual controls with the **`Margin`** property.

## The FlexLayout

The **`FlexLayout`** works like a **`StackLayout`**, since it arranges child visual elements vertically or horizontally, but the difference is that it is also able to wrap the child visual elements if there is not enough space in a single row or column. Code Listing 6 provides an example and shows how easy it is to work with this layout.

*Code Listing 6*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
```

```
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

        <FlexLayout Wrap="Wrap" JustifyContent="SpaceAround"
                    Direction="Row">
            <Label Text="This is a sample label in a page"
                   FlexLayout.AlignSelf="Center"/>
            <Button Text="Tap here to get things done"
                    FlexLayout.AlignSelf="Center" x:Name="Button1"/>
        </FlexLayout>
</ContentPage>
```

The **FlexLayout** exposes several properties, most of them common to other layouts, but the following are exclusive to **FlexLayout**, and certainly the most important to use to adjust its behavior:

- **Wrap**: A value from the **FlexWrap** enumeration that specifies if the **FlexLayout** content should be wrapped to the next row if there is not enough space in the first one. Possible values are **Wrap** (wraps to the next row), **NoWrap** (keeps the view content on one row), and **Reverse** (wraps to the next row in the opposite direction).
- **Direction**: A value from the **FlexDirection** enumeration that determines if the children of the **FlexLayout** should be arranged in a single row or column. The default value is **Row**. Other possible values are **Column**, **RowReverse**, and **ColumnReverse** (where **Reverse** means that child views will be laid out in the reverse order).
- **JustifyContent**: A value from the **FlexJustify** enumeration that specifies how child views should be arranged when there is extra space around them. There are self-explanatory values such as **Start**, **Center**, and **End**, as well other options such as **SpaceAround**, where elements are spaced with one unit of space at the beginning and end, and two units of space between them, so the elements and the space fill the line; and **SpaceBetween**, where child elements are spaced with equal space between units and no space at either end of the line, again so the elements and the space fill the line. The **SpaceEvenly** value causes child elements to be spaced so the same amount of space is set between each element as there is from the edges of the parent to the beginning and end elements.

You can specify the alignment of child views in the **FlexLayout** by assigning the **FlexLayout.AlignSelf** attached property with self-explanatory values such as **Start**, **Center**, **End**, and **Stretch**. For a quick understanding, you can take a look at Figure 26, which demonstrates how child views have been wrapped.

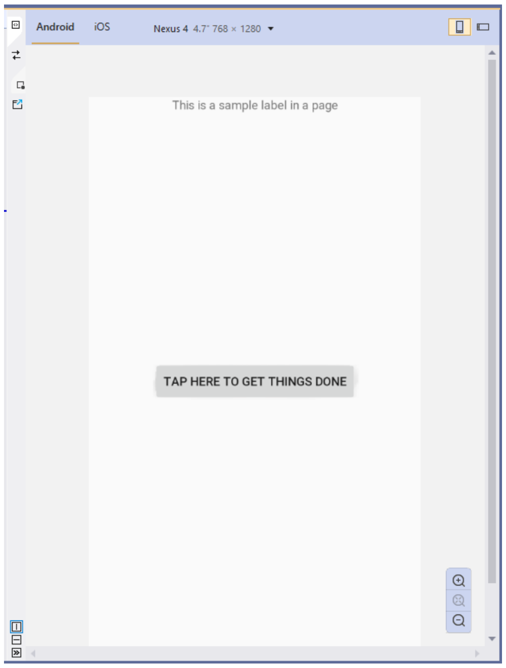


*Figure 26: Arranging visual elements with the FlexLayout*

If you change the **Wrap** property value to **NoWrap**, child views will be aligned on the same row, overlapping each other. The **FlexLayout** is therefore particularly useful for creating dynamic hierarchies of visual elements, especially when you do not know in advance the size of child elements.

## The Grid

The **Grid** is one of the easiest layouts to understand, and probably the most versatile. It allows you to create tables with rows and columns. In this way, you can define cells, and each cell can contain a control or another layout storing nested controls. The **Grid** is versatile in that you can just divide it into rows or columns, or both.

The following code defines a **Grid** that is divided into two rows and two columns:

```
<Grid>
   <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
   </Grid.RowDefinitions>
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>
</Grid>
```

**RowDefinitions** is a collection of **RowDefinition** objects, and the same is true for **ColumnDefinitions** and **ColumnDefinition**. Each item represents a row or a column within the **Grid**, respectively. You can also specify a **Width** or a **Height** property to delimit row and column dimensions; if you do not specify anything, both rows and columns are dimensioned at the maximum size available. When resizing the parent container, rows and columns are automatically rearranged.

The preceding code creates a table with four cells. To place controls in the **Grid**, you specify the row and column position via the **Grid.Row** and **Grid.Column** properties, known as attached properties, on the control. Attached properties allow for assigning properties of the parent container from the current visual element. The index of both is zero-based, meaning that **0** represents the first column from the left and the first row from the top. You can place nested layouts within a cell or a single row or column.

The code in Code Listing 7 shows how to nest a grid into a root grid with children controls.

***Tip:** `Grid.Row="0"` **and** `Grid.Column="0"` **can be omitted.***

*Code Listing 7*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Layouts.GridSample">
    <ContentPage.Content>
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition />
                <RowDefinition />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition />
                <ColumnDefinition />
            </Grid.ColumnDefinitions>
            <Button Text="First Button" />
            <Button Grid.Column="1" Text="Second Button"/>

            <Grid Grid.Row="1">
                <Grid.RowDefinitions>
                    <RowDefinition />
                    <RowDefinition />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition />
                </Grid.ColumnDefinitions>
                <Button Text="Button 3" />
                <Button Text="Button 4" Grid.Column="1" />
            </Grid>
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

Figure 27 shows the result of this code.

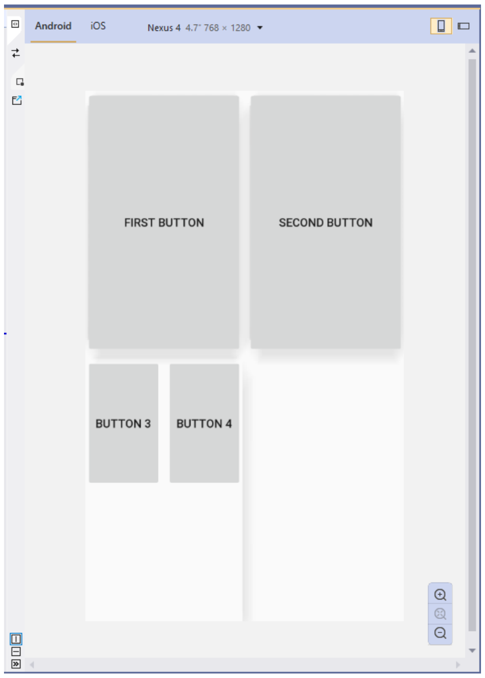


*Figure 27: Arranging visual elements with the Grid*

The **`Grid`** layout is very versatile and is also a good choice (when possible) in terms of performance.

## Spacing and proportions for rows and columns

You have fine-grained control over the size, space, and proportions of rows and columns. The **`Height`** and **`Width`** properties of the **`RowDefinition`** and **`ColumnDefinition`** objects can be set with values from the **`GridUnitType`** enumeration as follows:

- **`Auto`**: Automatically sizes to fit content in the row or column.
- **`Star`**: Sizes columns and rows as a proportion of the remaining space.
- **`Absolute`**: Sizes columns and rows with specific, fixed height and width values.

XAML has type converters for the **`GridUnitType`** values, so you simply pass no value for **`Auto`**, a **`*`** for **`Star`**, and the fixed numeric value for **`Absolute`**, such as:

```
<Grid.ColumnDefinitions>
   <ColumnDefinition />
   <ColumnDefinition Width="*"/>
   <ColumnDefinition Width="20"/>
```

```
</Grid.ColumnDefinitions>
```

### Introducing spans

In some situations, you might have elements that should occupy more than one row or column. In these cases, you can assign the **`Grid.RowSpan`** and **`Grid.ColumnSpan`** attached properties with the number of rows and columns a visual element should occupy.

## The AbsoluteLayout

The **`AbsoluteLayout`** container allows you to specify where exactly on the screen you want the child elements to appear, as well as their size and bounds. There are a few different ways to set the bounds of the child elements based on the **`AbsoluteLayoutFlags`** enumeration used during this process.

The **`AbsoluteLayoutFlags`** enumeration contains the following values:

- **`All`**: All dimensions are proportional.
- **`HeightProportional`**: Height is proportional to the layout.
- **`None`**: No interpretation is done.
- **`PositionProportional`**: Combines **`XProportional`** and **`YProportional`**.
- **`SizeProportional`**: Combines **`WidthProportional`** and **`HeightProportional`**.
- **`WidthProportional`**: Width is proportional to the layout.
- **`XProportional`**: **`X`** property is proportional to the layout.
- **`YProportional`**: **`Y`** property is proportional to the layout.

Once you have created your child elements, to set them at an absolute position within the container, you will need to assign the **`AbsoluteLayout.LayoutFlags`** attached property. You will also want to assign the **`AbsoluteLayout.LayoutBounds`** attached property to give the elements their bounds. Since Xamarin.Forms is an abstraction layer between Xamarin and the device-specific implementations, the positional values can be independent of the device pixels. This is where the **`LayoutFlags`** mentioned previously come into play. Code Listing 8 provides an example based on proportional dimensions and absolute position for child controls.

*Code Listing 8*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <AbsoluteLayout>
        <Label Text="First Label"
               AbsoluteLayout.LayoutBounds="0, 0, 0.25, 0.25"
               AbsoluteLayout.LayoutFlags="All" TextColor="Red"/>
        <Label Text="Second Label"
```

```
              AbsoluteLayout.LayoutBounds="0.20, 0.20, 0.25, 0.25"
              AbsoluteLayout.LayoutFlags="All" TextColor="Orange"/>
       <Label Text="Third Label"
              AbsoluteLayout.LayoutBounds="0.40, 0.40, 0.25, 0.25"
              AbsoluteLayout.LayoutFlags="All" TextColor="Violet"/>
       <Label Text="Fourth Label"
              AbsoluteLayout.LayoutBounds="0.60, 0.60, 0.25, 0.25"
              AbsoluteLayout.LayoutFlags="All" TextColor="Yellow"/>
   </AbsoluteLayout>
</ContentPage>
```

Figure 28 shows the result of the **`AbsoluteLayout`** example.

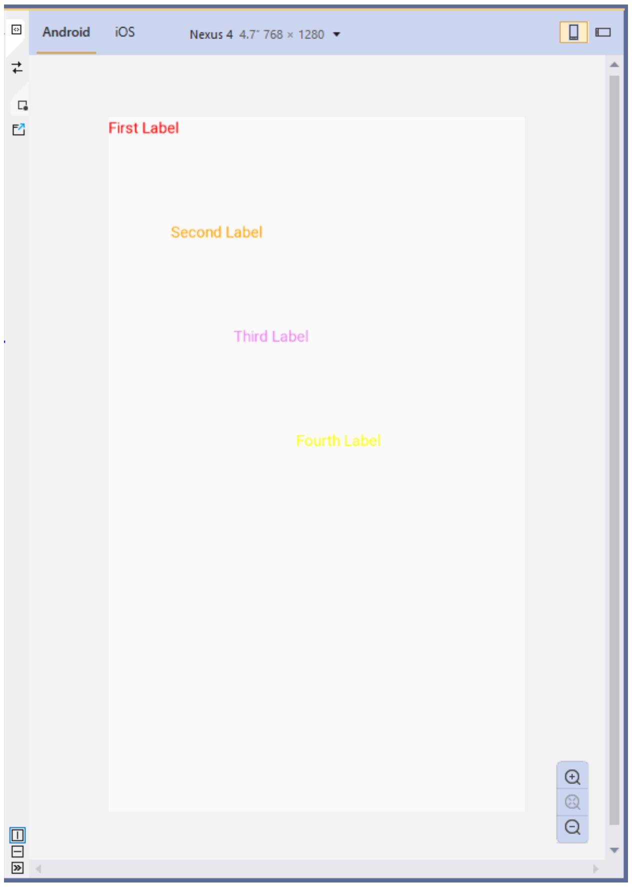


*Figure 28: Absolute positioning with AbsoluteLayout*

## The RelativeLayout

The **`RelativeLayout`** container provides a way to specify the location of child elements relative either to each other or to the parent control. Relative locations are resolved through a series of **`Constraint`** objects that define each particular child element's relative position to another. In XAML, **`Constraint`** objects are expressed through the **`ConstraintExpression`** markup extension, which is used to specify the location or size of a child view as a constant, or relative to a parent or other named view. Markup extensions are very common in XAML, and you will see many of them in Chapter 7 related to data binding, but discussing them in detail is beyond the scope here. The official documentation has a very detailed page on their syntax and implementation that I encourage you to read.

In the **`RelativeLayout`** class, there are properties named **`XConstraint`** and **`YConstraint`**. In the next example, you will see how to assign a value to these properties from within another XAML element, through attached properties. This is demonstrated in Code Listing 9, where you meet the **`BoxView`**, a visual element that allows you to draw a colored box. In this case, it's useful for giving you an immediate perception of how the layout is organized.

*Code Listing 9*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <RelativeLayout>
        <BoxView Color="Red" x:Name="redBox"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToPa
rent,
            Property=Height,Factor=.15,Constant=0}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=1,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.8,Constant=0}" />
        <BoxView Color="Blue"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToVi
ew,
            ElementName=redBox,Property=Y,Factor=1,Constant=20}"
        RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToVi
ew,
            ElementName=redBox,Property=X,Factor=1,Constant=20}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=.5,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.5,Constant=0}" />
    </RelativeLayout>
</ContentPage>
```

The result of Code Listing 9 is shown in Figure 29.

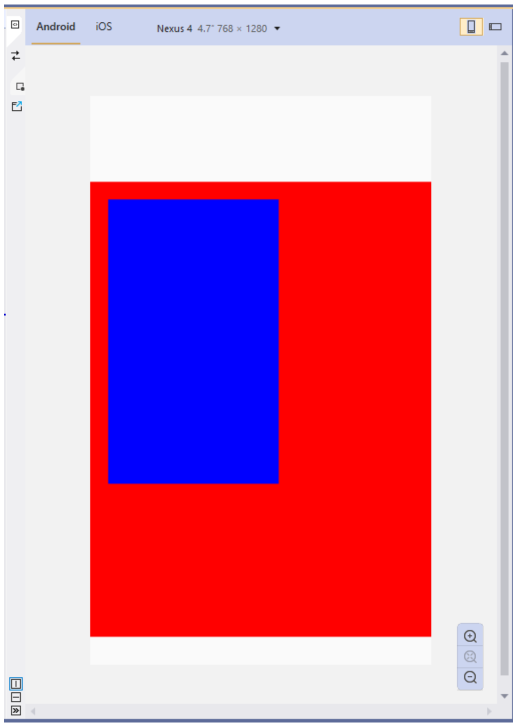


*Figure 29: Arranging visual elements with the RelativeLayout*

***Tip: The `RelativeLayout` container has relatively poor rendering performance, and the documentation recommends that you avoid this layout whenever possible, or at least avoid more than one `RelativeLayout` per page.***

## The ScrollView

The special layout **`ScrollView`** allows you to present content that cannot fit on one screen, and therefore should be scrolled. Its usage is very simple:

```
<ScrollView x:Name="Scroll1">
    <StackLayout>
        <Label Text="My favorite color:" x:Name="Label1"/>
        <BoxView BackgroundColor="Green" HeightRequest="600" />
    </StackLayout>
</ScrollView>
```

You basically add a layout or visual elements inside the **`ScrollView`** and, at runtime, the content will be scrollable if its area is bigger than the screen size. You can also decide whether to display the scroll bars through the **`HorizontalScrollbarVisibility`** and **`VerticalScrollbarVisibility`** properties that can be assigned with self-explanatory values such as **`Always`**, **`Never`**, and **`Default`**.

Additionally, you can specify the **Orientation** property (with values **Horizontal** or **Vertical**) to set the **ScrollView** to scroll only horizontally or only vertically. The reason the layout has a name in the sample usage is that you can interact with the **ScrollView** programmatically, invoking its **ScrollToAsync** method to move its position based on two different options.

Consider the following lines:

```
Scroll1.ScrollToAsync(0, 100, true);
Scroll1.ScrollToAsync(Label1, ScrollToPosition.Start, true);
```

In the first case, the content at 100 px from the top is visible. In the second case, the **ScrollView** moves the specified control at the top of the view and sets the current position at the control's position. Possible values for the **ScrollToPosition** enumeration are:

- **Center**: Scrolls the element to the center of the visible portion of the view.
- **End**: Scrolls the element to the end of the visible portion of the view.
- **MakeVisible**: Makes the element visible within the view.
- **Start**: Scrolls the element to the start of the visible portion of the view.

Note that you should never nest **ScrollView** layouts, and you should never include the **ListView** and **WebView** controls inside a **ScrollView** because they both already implement scrolling.

## The Frame

The **Frame** is a very special layout in Xamarin.Forms because it provides an option to draw a colored border around the visual element it contains, and optionally add extra space between the **Frame**'s bounds and the visual element. Code Listing 10 provides an example.

*Code Listing 10*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             x:Class="App1.MainPage">

    <Frame OutlineColor="Red" CornerRadius="3" HasShadow="True"
Margin="20">
        <Label Text="Label in a frame"
               HorizontalOptions="Center"
               VerticalOptions="Center"/>
    </Frame>
</ContentPage>
```

The **OutlineColor** property is assigned with the color for the border, the **CornerRadius** property is assigned with a value that allows you to draw circular corners, and the **HasShadow** property allows you to display a shadow. Figure 30 provides an example.

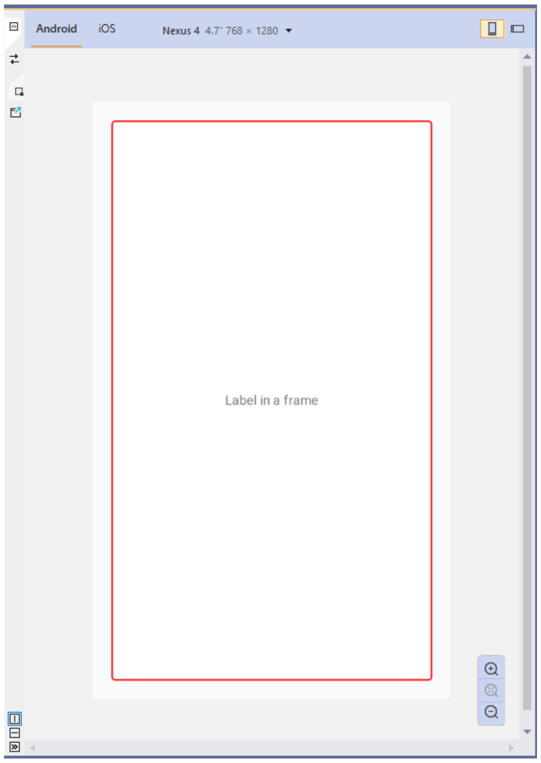


*Figure 30: Drawing a Frame*

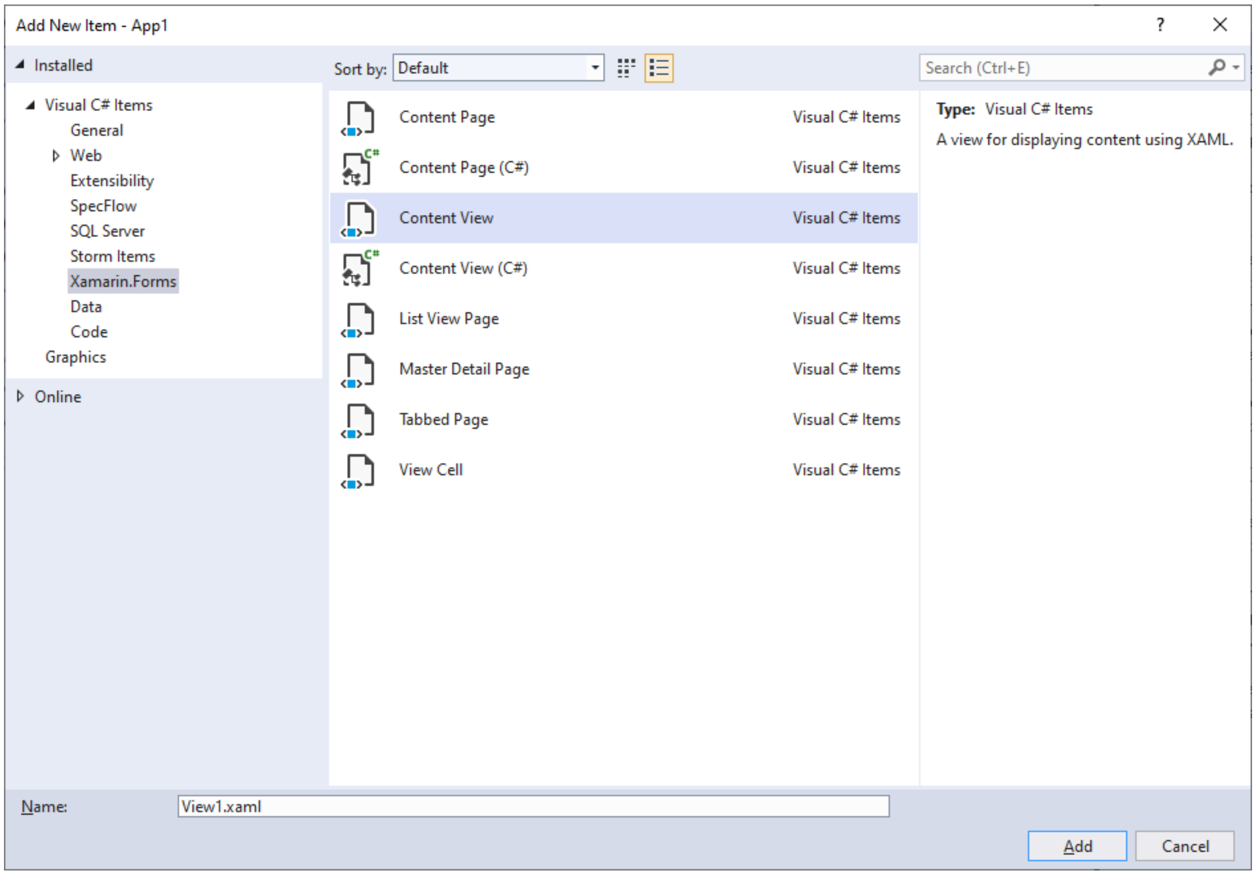
The **`Frame`** will be resized proportionally based on the parent container's size.

***Tip: In Visual Studio 2019, for properties of type `Color`, the XAML code editor shows a small inline box with a preview of the color itself.***

## The ContentView

The special container **`ContentView`** allows for aggregating multiple views into a single view and is useful for creating reusable, custom controls. Because the **`ContentView`** represents a stand-alone visual element, Visual Studio makes it easier to create an instance of this container with a specific item template.

In Solution Explorer, you can right-click the .NET Standard project name and then select **Add New Item**. In the **Add New Item** dialog box, select the **Xamarin.Forms** node, then the **Content View** item, as shown in Figure 31. Make sure you do not select the item called Content View (C#); otherwise, you will get a new class file that you will need to populate in C# code rather than XAML.



Sort by: Default

**Type:** Visual C# Items

A view for displaying content using XAML.

Name: View1.xaml

*Figure 31: Adding a ContentView*

When the new file is added to the project, the XAML editor shows basic content made of the **`ContentView`** root element and a **`Label`**. You can add multiple visual elements, as shown in Code Listing 11, and then you can use the **`ContentView`** as you would with an individual control or layout.

*Code Listing 11*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ContentView xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.View1">
  <ContentView.Content>
      <StackLayout>
          <Label Text="Enter your email address:" />
          <Entry x:Name="EmailEntry" />
      </StackLayout>
  </ContentView.Content>
</ContentView>
```

It is worth mentioning that visual elements inside a **ContentView** can raise and manage events and support data binding, which makes the **ContentView** very versatile and perfect for building reusable views.

## Styling the user interface with CSS

Xamarin.Forms allows you to style the user interface with cascading style sheets (CSS). If you have experience with creating content with HTML, you might find this feature very interesting.

***Note: CSS styles must be compliant with Xamarin.Forms in order to be consumed in mobile apps, since it does not support all CSS elements. For this reason, this feature should be considered a complement to XAML, not a replacement. Before you decide to make serious styling with CSS in your mobile apps, make sure you read the documentation for further information about what is available and supported.***

There are three options to consume CSS styles in a Xamarin.Forms project: two in XAML, and one in C# code.

### Defining CSS styles as a XAML resource

***Note: Examples in this section are based on the*** `ContentPage` ***object since you have not read about other pages yet, but the concepts apply to all pages deriving from*** `Page`***. Chapter 6 will describe in detail all the available pages in Xamarin.Forms.***

The first way you can use CSS styles in Xamarin.Forms is by defining a **StyleSheet** object within the resources of a page, like in the following code snippet:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Layouts.CSSsample">
    <ContentPage.Resources>
        <ResourceDictionary>
            <StyleSheet>
                <![CDATA[
^contentpage {
background-color: lightgray; }
stacklayout {
margin: 20; }
]]>
            </StyleSheet>
        </ResourceDictionary>
    </ContentPage.Resources>
</ContentPage>
```

In this scenario, the CSS content is enclosed within a **CDATA** section.

***Note: Names of visual elements inside a CSS style must be lowercase.***

For each visual element, you supply property values in the form of key/value pairs. The syntax requires the visual element name and, enclosed within brackets, the property name followed by a colon, and the value followed by a semicolon, such as **`stacklayout { margin: 20; }`**. Notice how the root element, **`contentpage`** in this case, must be preceded by the **^** symbol. You do not need to do anything else, as the style will be applied to all the visual elements specified in the CSS.

## Consuming CSS files in XAML

The second available option to consume a CSS style in XAML is from an existing .css file. First, you need to add your .css file to the Xamarin.Forms project (either .NET Standard or shared project) and set its **`BuildAction`** property as **`EmbeddedResource`**. The next step is to add a **`StyleSheet`** object to a **`ContentPage`**’s resources and assign its **`Source`** property with the .css file name, as follows:

```
    <ContentPage.Resources>
        <ResourceDictionary>
            <StyleSheet Source="/mystyle.css"/>
        </ResourceDictionary>
    </ContentPage.Resources>
```

Obviously, you can organize your .css files into subfolders; for example, the value for the **`Source`** property could be **`/Assets/mystyle.css`**.

## Consuming CSS styles in C# code

The last option you have to consume CSS styles in Xamarin.Forms is using C# code. You can create a CSS style from a string (through a **`StringReader`** object), or you can load an existing style from a .css file, but in both cases, the key point is that you still need to add the style to a page’s resources. The following code snippet demonstrates the first scenario, where a CSS style is created from a string and added to the page’s resources:

```
using (var reader =
       new StringReader
           ("^contentpage { background-color: lightgray; }
             stacklayout { margin: 20; }"))
{
       // "this" represents a page
       // StyleSheet requires a using Xamarin.Forms.StyleSheets directive
       this.Resources.Add(StyleSheet.FromReader(reader));
}
```

For the second scenario, loading the content of a CSS style from an existing file, an example is provided by the following code snippet:

```
var styleSheet = StyleSheet.FromAssemblyResource(IntrospectionExtensions.
                 GetTypeInfo(typeof(Page1)).Assembly,
                 "Project1.Assets.mystyle.css");
this.Resources.Add(styleSheet);
```

The second snippet is a bit more complex, since the file is loaded via reflection (and in fact, it requires a **`using System.Reflection`** directive in order to import the **`IntrospectionExtensions`** object). Notice how you provide the file name including the project name (**`Project1`**) and the subfolder (if any) name that contains the .css file.

## Chapter summary

Mobile apps require dynamic user interfaces that can automatically adapt to the screen size of different device form factors. In Xamarin.Forms, creating dynamic user interfaces is possible through a number of so-called layouts.

The **`StackLayout`** allows you to arrange controls near one another both horizontally and vertically. The **`FlexLayout`** does the same, but it is also capable of wrapping visual elements. The **`Grid`** allows you to arrange controls within rows and columns; the **`AbsoluteLayout`** allows you to give controls an absolute position; the **`RelativeLayout`** allows you to arrange controls based on the size and position of other controls or containers; the **`ScrollView`** layout allows you to scroll the content of visual elements that do not fit in a single page; the **`Frame`** layout allows you to draw a border around a visual element; and the **`ContentView`** allows you to create reusable views.

In the last part of the chapter, you saw how you can style visual elements using CSS stylesheets, in both XAML and imperative code, but these must be compliant to Xamarin.Forms and should only be considered as a complement to XAML, and not a replacement.

Now that you have a basic knowledge of layouts, it’s time to discuss common controls in Xamarin.Forms that allow you to build the functionalities of the user interface, arranged within the layouts you learned in this chapter.

# Chapter 5 Xamarin.Forms Common Controls

Xamarin.Forms ships with a rich set of common controls that you can use to build cross-platform user interfaces easily, and without the need for the complexity of platform-specific features. As you can imagine, the benefit of these common controls is that they run on Android, iOS, and Windows from the same codebase. In this chapter, you'll learn about common controls, their properties, and their events. Other controls will be introduced in Chapter 7, especially controls whose purpose is displaying lists of data.

***Note: In order to follow the examples in this chapter, either create a brand new Xamarin.Forms solution (the name is up to you) or open the solution in the Chapter5 folder of the companion code repository. If you go for the first option, every time a new control is discussed, just clean the content of the root `ContentPage` object in the XAML file and remove any C# code specific to a single control or add a new file of type `ContentPage` to the project.***

## Understanding the concept of view

In Xamarin.Forms, a *view* is the building block of any mobile application. Put succinctly, a view is a control, and it represents what you would call a widget in Android, a view in iOS, and a control in Windows. Views derive from the **`Xamarin.Forms.View`** class. Actually, from a technical perspective, layouts are views themselves and derive from **`Layout`**, an intermediate object in the hierarchy that derives from **`View`** and includes a **`Children`** property, allowing you to add multiple visual elements to the layout itself.

The concept of view is also important from the terminology perspective. In fact, in Xamarin.Forms and its documentation, you will more often find the word *view* than *control*. From now on, I will be using both view and control interchangeably, but remember that documentation and tutorials often refer to views.

### Views' common properties

Views share a number of properties that are important for you to know in advance. These are summarized in Table 4.

*Table 4: Views' common properties*

| Property | Description |
|---|---|
| **`HorizontalOptions`** | Same as Table 2. |
| **`VerticalOptions`** | Same as Table 2. |

| Property | Description |
|---|---|
| `HeightRequest` | Of type **`double`**, gets or sets the height of a view. |
| `WidthRequest` | Of type **`double`**, gets or sets the width of a view. |
| `IsVisible` | Of type **`bool`**, determines whether a control is visible on the user interface. |
| `IsEnabled` | Of type **`bool`**, allows enabling or disabling a control, keeping it visible on the UI. |
| `GestureRecognizers` | A collection of **`GestureRecognizer`** objects that enable touch gestures on controls that do not directly support touch. These will be discussed later in this chapter. |

***Tip: Controls also expose the*** *`Margin`* ***property described in Table 3.***

If you wish to change the width or height of a view, remember that you need the **`WidthRequest`** and **`HeightRequest`** properties, instead of **`Height`** and **`Width`**, which are read-only and return the current height and width.

## Introducing common views

This section provides a high-level overview of common views in Xamarin.Forms, and their most utilized properties. Remember to add the official documentation about the user interface to your bookmarks for a more detailed reference.

### User input with the Button

The **`Button`** control is certainly one of the most-used controls in every user interface. You already saw a couple examples of the **`Button`** previously, but here is a quick summary. This control exposes the properties summarized in Table 5, and you declare it like this:

```
<Button x:Name="Button1" Text="Tap here" TextColor="Orange" BorderColor="Red"
        BorderWidth="2" CornerRadius="2" Clicked="Button1_Clicked"/>
```

*Table 5: Button properties*

| Property | Description |
|---|---|
| **`Text`** | The text in the button. |
| **`TextColor`** | The color of the text in the button. |
| **`BorderColor`** | Draws a colored border around the button. |
| **`BorderWidth`** | The width of the border around the button. |
| **`CornerRadius`** | The radius of the edges around the button. |
| **`Image`** | An optional image to be set near the text. |

Notice how you can specify a name with **`x:Name`** so that you can interact with the button in C#, which is the case when you set the **`Clicked`** event with a handler. This control also exposes the **`Font`**, **`FontFamily`**, and **`FontAttributes`** properties, whose behavior is discussed in the next section about text.

## Working with text: Label, Entry, and Editor

Displaying text and requesting input from the user in the form of text is extremely common in every mobile app. Xamarin.Forms offers the **`Label`** control to display read-only text, and the **`Entry`** and **`Editor`** controls to receive text. The **`Label`** control has some useful properties, as shown in the following XAML:

```
<Label Text="Displaying some text" LineBreakMode="WordWrap"
   TextColor="Blue" XAlign="Center" YAlign="Center"/>
```

**`LineBreakMode`** allows you to truncate or wrap a long string and can be assigned a value from the **`LineBreakMode`** enumeration. For example, **`WordWrap`** splits a long string into multiple lines proportionate to the available space. If not specified, **`NoWrap`** is the default. **`XAlign`** and **`YAlign`** specify the horizontal and vertical alignment for the text. The **`Entry`** control instead allows you to enter a single line of text, and you declare it as follows:

```
<Entry x:Name="Entry1" Placeholder="Enter some text..."
   TextColor="Green" Keyboard="Chat" Completed="Entry1_Completed"/>
```

The **`Placeholder`** property lets you display specific text in the entry until the user types something. It is useful for explaining the purpose of the text box. When the user taps the **`Entry`**, the on-screen keyboard is displayed. The appearance of the keyboard can be controlled via the **`Keyboard`** property, which allows you to display the most appropriate keys, depending on the **`Entry`**'s purpose. Supported values are **`Chat`**, **`Email`**, **`Numeric`**, **`Telephone`**, and **`Number`**. If **`Keyboard`** is not assigned, **`Default`** is assumed.

Additionally, **Entry** exposes the **MaxLength** property, which allows you to set the max length of the text the user can enter. This control also exposes two events: **Completed**, which is fired when the users finalize the text by tapping the Return key, and **TextChanged**, which is fired at every keystroke.

You provide event handlers the usual way, as follows:

```
private void Entry1_Completed(object sender, EventArgs e)
{
    // Entry1.Text contains the full text
}
```

**Entry** also provides the **IsPassword** property to mask the **Entry**'s content, which you use when the user must enter a password. Another very useful property is **ReturnType**, whose value is of type **ReturnType**, which allows you to specify the text to be displayed in the Enter key of the keyboard. Possible values are **Done**, **Go**, **Next**, **Search**, **Send**, and **Default** (which sets the default text for each platform). The combination of the **Label** and **Entry** controls is visible in Figure 32.

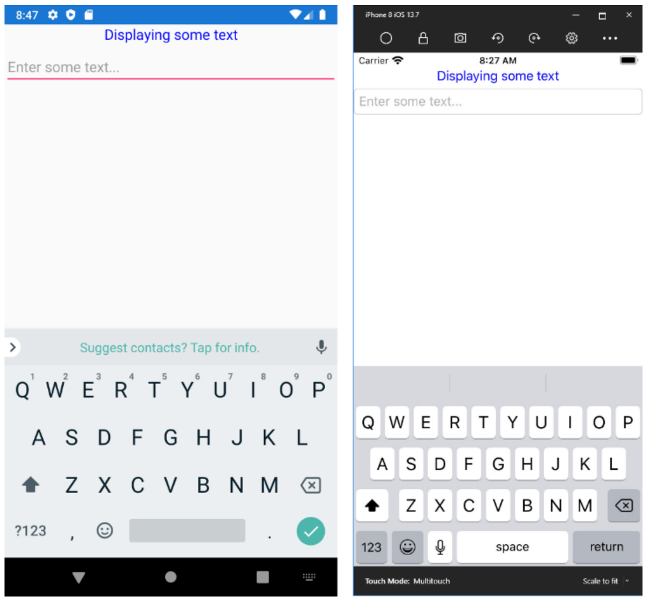


*Figure 32: Label and Entry controls*

The **Editor** control is very similar to **Entry** in terms of behavior, events, and properties, but it allows for entering multiple lines of text. For example, if you place the editor inside a layout, you can set its **HorizontalOptions** and **VerticalOptions** properties with **Fill** so that it will take all the available space in the parent. Both the **Entry** and **Editor** views expose the **IsSpellChedkedEnabled** property that, as the name implies, enables spell check over the entered text.

## Formatted strings and bindable spans

The **Label** control exposes the **FormattedText** property, which can be used to implement more sophisticated string formatting. This property is of type **FormattedString**, an object that is populated with a collection of **Span** objects, each representing a part of the formatted string. The following snippet provides an example:

```
<Label
   HorizontalOptions="Center"
   VerticalOptions="CenterAndExpand">
    <Label.FormattedText>
        <FormattedString>
            <FormattedString.Spans>
                <Span FontSize="Large" FontAttributes="Bold"
                      ForegroundColor="Red"
                      Text="Xamarin.Forms Succinctly" />
                <Span FontSize="Medium" FontAttributes="Italic"
                      ForegroundColor="Black" Text="3rd edition" />
                <Span FontSize="Small" FontAttributes="Bold"
                      ForegroundColor="Blue"
                      Text="by Alessandro Del Sole" />
            </FormattedString.Spans>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

You can basically use **Span** objects to format specific paragraphs or sentences by setting most of the properties already exposed by the **Label** control, such as **Text**, **Font**, **FontFamily**, **FontSize**, **FontAttributes**, **ForegroundColor**, and **BackgroundColor**.

## Managing fonts

Controls that display some text (including the **Button**) or that wait for user input through the keyboard also expose some properties related to fonts, such as **FontFamily**, **FontAttributes**, and **FontSize**. **FontFamily** specifies the name of the font you want to use. **FontAttributes** displays text as **Italic** or **Bold** and, if not specified, **None** is assumed.

With **FontSize**, you can specify the font size with either a numeric value or with a so-called *named size*, based on the **Micro**, **Small**, **Medium**, and **Large** values from the **NamedSize** enumeration. With this enumeration, Xamarin.Forms chooses the appropriate size for the current platform. For instance, the following two options are allowed to set the font size:

```
<Label Text="Some text" FontSize="72"/>
<Label Text="Some text" FontSize="Large"/>
```

Unless you are writing an app for a single platform, I recommend you avoid using numeric values—use the named size instead. You can also manage the spacing between individual characters in a string by setting the **CharacterSpacing** property, of type **double**, whose default value is zero.

Another useful property is **`TextTransform`**, which you can use to automatically convert a string into uppercase or lowercase. Allowed values are **`Uppercase`**, **`Lowercase`**, **`Default`**, and **`None`**. By default, strings are not transformed. It is also possible to apply underline and strikethrough decorations to a string by assigning the **`TextDecorations`** property with **`Underline`** and **`Strikethrough`** (the default is **`None`**).

### Using custom fonts

A common requirement with modern apps is to use custom fonts. These can be available through official design channels, such as Google Material, or produced by professional designers. Xamarin.Forms supports both .ttf and .otf file formats. In order to use custom fonts, follow these steps:

1. Add the .ttf or .otf files into each platform project's resource folder: Assets for Android and UWP, Resources for iOS.
2. Set the Build Action property of each file to: **`AndroidResource`** for Android, **`BundleResource`** for iOS, and **`EmbeddedResource`** for UWP.
3. Assign the **`FontFamily`** property of your view with the name of the font.

For iOS and UWP, **`FontFamily`** just takes the font name, as follows:

```
<Label Text="Some text" FontFamily="MyFont.ttf"/>
```

For Android, you also need to specify the font name without the extension, as follows:

```
<Label Text="Some text" FontFamily="MyFont.ttf#MyFont"/>
```

In Chapter 9 you will learn how to assign the same property with different values depending on the operating system.

## Working with dates and time: DatePicker and TimePicker

Another common requirement in mobile apps is working with dates and time: Xamarin.Forms provides the **`DatePicker`** and **`TimePicker`** views for that. On each platform, these are rendered with the corresponding date and time selectors. **`DatePicker`** exposes the **`Date`**, **`MinimumDate`**, and **`MaximumDate`** properties that represent the selected/current date, the minimum date, and the maximum date, respectively, all of type **`DateTime`**. It exposes an event called **`DateSelected`**, which is raised when the user selects a date. You can handle this to retrieve the value of the **`Date`** property. The view can be declared as follows:

```
<DatePicker x:Name="DatePicker1" MinimumDate="07/17/2021"
            MaximumDate="12/31/2021"
            DateSelected="DatePicker1_DateSelected"/>
```

And then in the code-behind, you can retrieve the selected date, like this:

```
private void DatePicker1_DateSelected(object sender, DateChangedEventArgs e)
{
    DateTime selectedDate = e.NewDate;
}
```

The **`DateChangedEventArgs`** object stores the selected date in the **`NewDate`** property and the previous date in the **`OldDate`** property. Figure 33 shows the **`DatePicker`** on Android and iOS simulators.

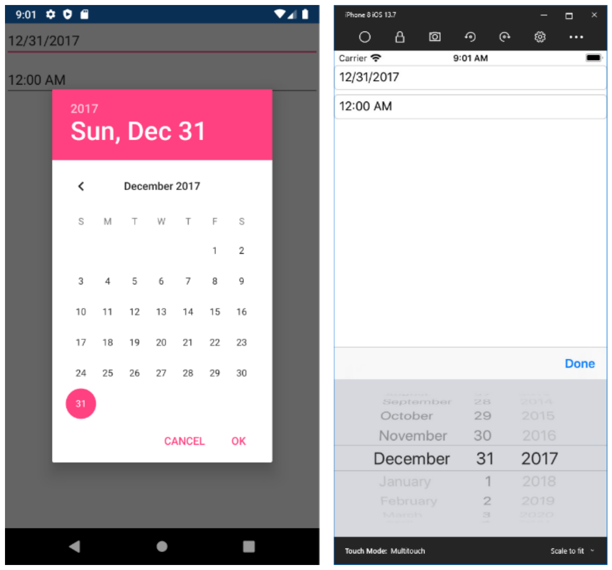


*Figure 33: The DatePicker in action*

The **`TimePicker`** exposes a property called **`Time`**, of type **`TimeSpan`**, but it does not expose a specific event for time selection, so you need to use the **`PropertyChanged`** event. In terms of XAML, you declare a **`TimePicker`** like this:

```
<TimePicker x:Name="TimePicker1"
            PropertyChanged="TimePicker1_PropertyChanged"/>
```

Then, in the code-behind, you need to detect changes on the **`Time`** property as follows:

```
private void TimePicker1_PropertyChanged(object sender,
    System.ComponentModel.PropertyChangedEventArgs e)
{
    if(e.PropertyName == TimePicker.TimeProperty.PropertyName)
    {
        TimeSpan selectedTime = TimePicker1.Time;
    }
}
```

**`TimeProperty`** is a dependency property, a concept that will be discussed in Chapter 7. Figure 34 shows the **`TimePicker`** in action.

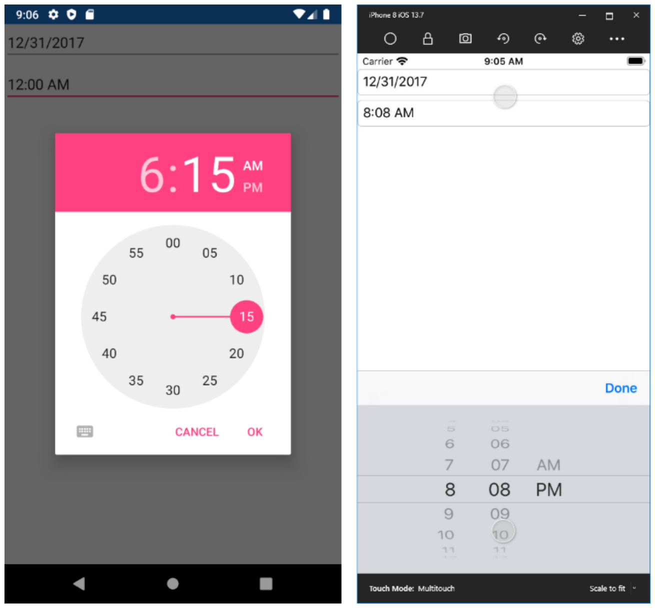


*Figure 34: The TimePicker in action*

***Tip: You can also assign a date or time to pickers in the C# code-behind, for example, in the constructor of the page that declares them.***

## Displaying HTML contents with WebView

The **`WebView`** control allows for displaying HTML contents, including webpages and static HTML markup. This control exposes the **`Navigating`** and **`Navigated`** events that are raised when navigation starts and completes, respectively. The real power is in its **`Source`** property, of type **`WebViewSource`**, which can be assigned with a variety of content, such as URIs or strings containing HTML markup. For example, the following XAML opens the specified website:

```
<WebView x:Name="WebView1" Source="https://www.xamarin.com"/>
```

The following example shows instead how you can assign the **`Source`** property with a string:

```
WebView1.Source = "<div><h1>Header</h1></div>";
```

For dynamic sizing, a better option is enclosing the **`WebView`** inside a **`Grid`** layout. If you instead use the **`StackLayout`**, you need to supply height and width explicitly. When you browse contents on the internet, you need to enable the internet permission in the Android manifest, and the internet (client) permission in the UWP manifest. Figure 35 shows how the **`WebView`** appears.

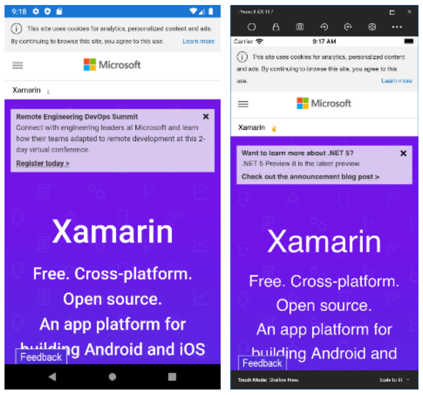

Microsoft
Xamarin
Free. Cross-platform. Open source. An app platform for building Android and iOS
Feedback

*Figure 35: Displaying HTML contents with WebView*

If the webpage you display allows you to browse other pages, you can leverage the built-in **`GoBack`** and **`GoForward`** methods, together with the **`CanGoBack`** and **`CanGoForward`** Boolean properties to programmatically control navigation between webpages.

***Tip: If you need to implement navigation to URLs, it might be worth considering the deep linking feature.***

### App Transport Security in iOS

Starting with iOS 9, Apple introduced some restrictions in accessing networked resources, including websites, enabling navigation only through the HTTPS protocol by default. This feature is known as App Transport Security (ATS). ATS can be controlled in the iOS project properties, and it allows for introducing some exceptions, because you might need to browse HTTP contents despite the restrictions. More details about ATS and exceptions are available in the documentation; however, remember that if the **`WebView`** shows no content on iOS, the reason might be ATS.

## Implementing value selection: Switch, Slider, Stepper

Xamarin.Forms offers a number of controls for user input based on selecting values. The first of them is the **`Switch`**, which provides a toggled value and is useful for selecting values such as true or false, on or off, and enabled or disabled. It exposes the **`IsToggled`** property, which turns the switch on when **`true`**, and the **`Toggled`** event, which is raised when the user changes the switch position. This control has no built-in label, so you need to use it in conjunction with a **`Label`**, as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Enable data plan"/>
   <Switch x:Name="Switch1" IsToggled="True" Toggled="Switch1_Toggled"
           Margin="5,0,0,0"/>
</StackLayout>
```

The **`Toggled`** event stores the new value in the **`ToggledEventArgs`** object that you use as follows:

```
private void Switch1_Toggled(object sender, ToggledEventArgs e)
{
    bool isToggled = e.Value;
}
```

You can also change the color for the selector when it's turned on, assigning the **OnColor** property with the color of your choice. Before Xamarin.Forms 3.1, you had to create a custom renderer to achieve this, so it is a very useful addition. The **Slider** allows the input of a linear value. It exposes the **Value**, **Minimum**, and **Maximum** properties, all of type **double**, which represent the current value, minimum value, and maximum value. Like the **Switch**, it does not have a built-in label, so you can use it together with a **Label** as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Select your age: "/>
   <Slider x:Name="Slider1" Maximum="85" Minimum="13" Value="30"
    ValueChanged="Slider1_ValueChanged"/>
</StackLayout>
```

***Tip: Surprisingly, if you write the*** `Minimum` ***before the*** `Maximum`***, a runtime error will occur. So, for both the*** `Slider` ***and the*** `Stepper`***, the order matters.***

The **ValueChanged** event is raised when the user moves the selector on the **Slider**, and the new value is sent to the **NewValue** property of the **ValueChangedEventArgs** object you get in the event handler.

The last control is the **Stepper**, which allows the supplying of discrete values with a specified increment. It also allows the specifying of minimum and maximum values. You use the **Value**, **Increment**, **Minimum**, and **Maximum** properties of type **double** as follows:

```
<StackLayout Orientation="Horizontal">
   <Label Text="Select your age: "/>
   <Stepper x:Name="Stepper1" Increment="1" Maximum="85" Minimum="13"
            Value="30" ValueChanged="Stepper1_ValueChanged"/>
   <Label x:Name="StepperValue"/>
</StackLayout>
```

Notice that both the **Stepper** and **Slider** only provide a way to increment and decrement a value, so it is your responsibility to display the current value, for example, with a **Label** that you can handle through the **ValueChanged** event. The following code demonstrates how to accomplish this with the **Stepper**:

```
private void Stepper1_ValueChanged(object sender, ValueChangedEventArgs e)
{
    StepperValue.Text = e.NewValue.ToString();
}
```

Figure 36 shows a summary of all the aforementioned views.

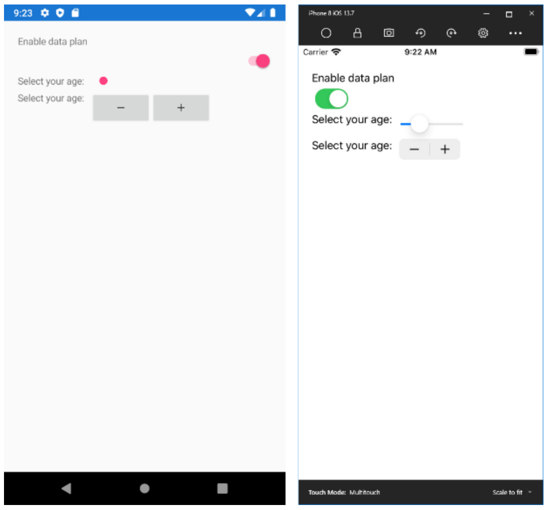


*Figure 36: A summary view of the Switch, Slider, and Stepper controls*

## Introducing the SearchBar

One of the nicest views in Xamarin.Forms, the **SearchBar** shows a native search box with a search icon that users can tap. This view exposes the **SearchButtonPressed** event. You can handle this event to retrieve the text the user typed in the box and then perform your search logic, for example, by executing a LINQ query against an in-memory collection or filtering data from the table of a local database.

It also exposes the **TextChanged** event, which is raised at every keystroke, and the **Placeholder** property, which allows you to specify a placeholder text like the same-named property of the **Entry** control. You declare it as follows:

```
<SearchBar x:Name="SearchBar1" Placeholder="Enter your search key..."
           SearchButtonPressed="SearchBar1_SearchButtonPressed"/>
```

Figure 37 shows an example.

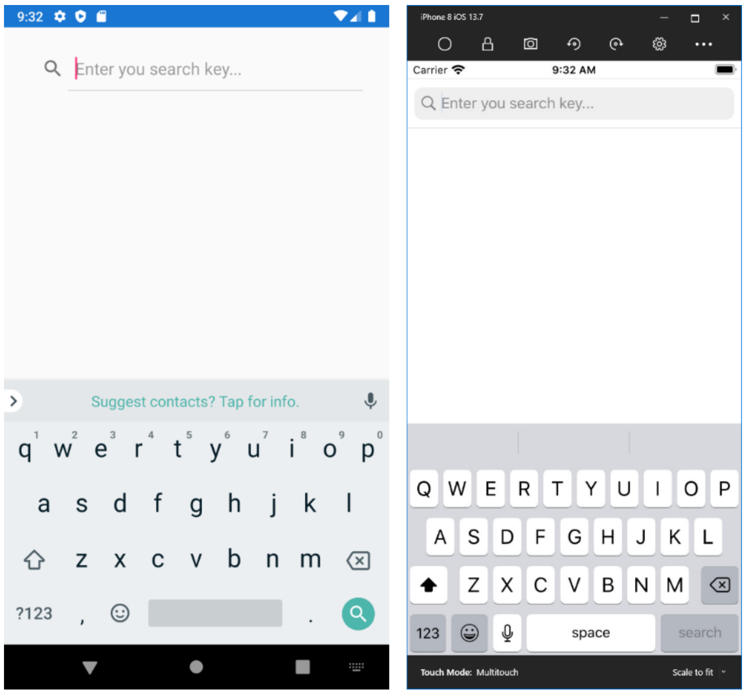


*Figure 37: The SearchBar view*

In Chapter 7, you will learn how to display lists of items through the **`ListView`** control. The **`SearchBar`** can be a good companion in that you can use it to filter a list of items based on the search key the user entered.

## Long-running operations: ActivityIndicator and ProgressBar

In some situations, your app might need to perform potentially long-running operations, such as downloading content from the internet or loading data from a local database. In such situations, it is a best practice to inform the user that an operation is in progress. This can be accomplished with two views, the **`ActivityIndicator`** or the **`ProgressBar`**. The latter exposes a property called **`Progress`**, of type **`double`** and a **`ProgressColor`** property, of type **`Color`**, that allows you to change the color of the progress indicator. This control is not used very often, because it implies you are able to calculate the amount of time or data needed to complete an operation.

The **`ActivityIndicator`** instead shows a simple, animated indicator that is displayed while an operation is running, without the need to calculate its progress. It is enabled by setting its **`IsRunning`** property to **`true`**; you might also want to make it visible only when running, done by assigning **`IsVisible`** with **`true`**. You typically declare it in XAML as follows:

```
<ActivityIndicator x:Name="ActivityIndicator1" />
```

Then, in the code-behind, you can control it as follows:

```
// Starting the operation...
ActivityIndicator1.IsVisible = true;
ActivityIndicator1.IsRunning = true;
```

```
// Executing the operation...

// Operation completed
ActivityIndicator1.IsRunning = false;
ActivityIndicator1.IsVisible = false;
```

As a personal suggestion, I recommend you always set both **`IsVisible`** and **`IsRunning`**. This will help you keep consistent behavior across platforms. Figure 38 shows an example.

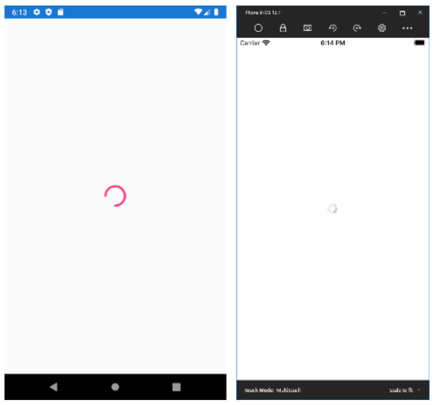


*Figure 38: The ActivityIndicator shows that an operation is in progress*

***Tip:* `Page` *objects, such as the* `ContentPage`*, expose a property called* `IsBusy` *that enables an activity indicator when assigned with* `true`*. Depending on your scenario, you might also consider this option.***

## Working with images

Using images is crucial in mobile apps since they both enrich the look and feel of the user interface and enable apps to support multimedia content. Xamarin.Forms provides an **`Image`** control you can use to display images from the internet, local files, and embedded resources. Displaying images is really simple, while understanding how you load and size images is more complex, especially if you have no previous experience with XAML and dynamic user interfaces. You declare an **`Image`** as follows:

```
<Image Source="https://cdn.syncfusion.com/content/images/downloads/ebook/eboo
k-cover/2018-bronze-xamarin-forms-succinctly.png" Aspect="AspectFit"/>
```

As you can see, you assign the **`Source`** property with the image path, which can be a URL or the name of a local file or resource. **`Source`** can be assigned either in XAML or in code-behind. You will assign this property in C# code when you need to assign the property at runtime. This property is of type **`ImageSource`** and, while XAML has a type converter for it, in C# you need to use specific methods, depending on the image source: **`FromFile`** requires a file path that can be resolved on each platform, **`FromUri`** requires a **`System.Uri`** object, and **`FromResource`** allows you to specify an image in the embedded app resources.

***Note: Each platform has its own way of working with local images and embedded resources, which requires further explanation. Because this goes beyond the scope of this ebook, I strongly recommend you read the official documentation, which also explains how to manage images for different purposes on iOS, Android, and Windows.***

The **`Aspect`** property determines how to size and stretch an image within the bounds it is being displayed in. It requires a value from the **`Aspect`** enumeration:

- **`Fill`**: Stretches the image to fill the display area completely and exactly. This may result in the image being distorted.
- **`AspectFill`**: Clips the image so that it fills the display area while preserving the aspect.
- **`AspectFit`**: Letterboxes the image (if required) so that the entire image fits into the display area, with blank space added to the top, bottom, or sides, depending on whether the image is wide or tall.

You can also set the **`WidthRequest`** and **`HeightRequest`** properties to adjust the size of the **`Image`** control. Figure 39 shows an example.

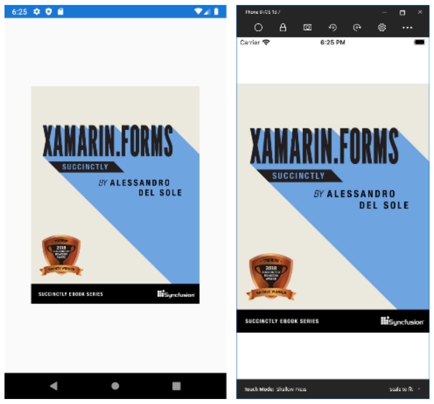


*Figure 39: Displaying images with the Image view*

Supported image formats are .jpg, .png, .gif, .bmp, and .tif. Working with images also involves icons and splash screens, which are totally platform-dependent, and therefore require you to read the official documentation. Also, for enhanced 2D graphics, you might want to consider taking a look at the SkiaSharp library, a portable library that works great with Xamarin.Forms and is powered by Google's Skia library.

## Playing audio and videos

Xamarin.Forms provides a view called **MediaElement**, which allows for playing audio and videos in your applications. This is a recent addition, but the demand for a native media player has always been very high. Actually, during its development, the **MediaElement** control was included in the Xamarin.Forms code base, but now it has been moved to the Xamarin Community Toolkit library, so you will need to install the Xamarin.CommunityToolkit NuGet package into your project. The **MediaElement** can play media contents from a remote URI, from the local library, from media files embedded inside the app resources, and from local folders.

For the sake of simplicity, the next examples will be based on a free, public Microsoft video about Xamarin that can be streamed online. The reason is that working with local videos requires knowledge of features like resources and native API access, which will be discussed later in the book. Additionally, only the most important properties, methods, and events will be discussed.

In its simplest form, you declare a **MediaElement** as follows:

```
<MediaElement ShowsPlaybackControls="True"

                              Source="https://sec.ch9.ms/ch9/5d93/a1eab4bf-3288-
4faf-81c4-294402a85d93/XamarinShow_mid.mp4"/>
```

The **Source** property contains the URI of the media file. As you can learn through the official documentation, local files are also represented by URIs that start with the **ms-appx:///** or **ms-appdata:///** prefixes. The **ShowsPlaybackControls** property allows you to avoid the need to create playback controls manually and will make the **MediaElement** use the playback control of each native platform.

You certainly have the option to create your custom controls and use data binding to provide a different look and feel to your player, but this is out of the scope of this chapter. You can make media start automatically by setting the **AutoPlay** property with **true**, and you can control the media volume using the **Volume** property, of type **double**. Its value must be between 0 and 1.

Like for the **Image** view, the **MediaElement** also exposes the **Aspect** property, which controls the stretching of the video with exactly the same values (**Fill**, **AspectFill**, **AspectFit**). The **Duration** property, of type **TimeSpan?**, returns the duration of the currently opened media, while the **Position** property, of type **TimeSpan**, returns the current progress over the duration. Figure 40 shows the **MediaElement** in action as declared above.

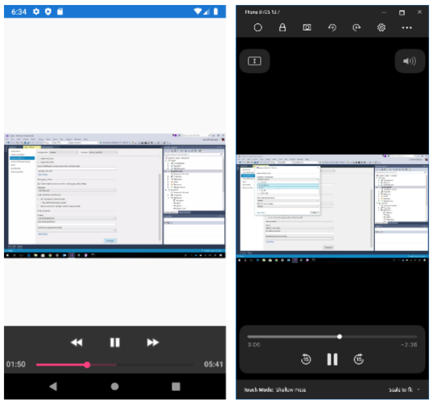


*Figure 40: Playing videos with the MediaElement*

The **`MediaElement`** view also exposes self-explanatory methods such as **`Play`**, **`Stop`**, and **`Pause`** that you can invoke in your C# code to manually control the media file. Additionally, and among the others, this view exposes events like:

- **`MediaOpened`**: Raised when the media stream has been validated and is ready for playing.
- **`MediaEnded`**: Raised when the media stream reaches its end.
- **`MediaFailed`**: Raised when an error occurs on the media source.

The **`MediaElement`** is at the same time a very versatile view in its simplest form, and a completely customizable view for high-quality media playing designs—but to get the most out of customizations, you need to first learn what comes in the next chapters.

> ***Tip: When you learn about data binding in Chapter 7, you will gain the necessary knowledge to build a great custom media player, with your own controls and features. For example, you could bind a `Slider` view to the `Volume` property, and another `Slider` to the `Position` property.***

## Introducing gesture recognizers

Views such as **`Image`** and **`Label`** do not include support for touch gestures natively, but sometimes you might want to allow users to tap a picture or text to perform an action such as navigating to a page or website. The **`Xamarin.Forms.Gestures`** namespace has classes that allow you to leverage gesture recognizers to add touch support to views that do not include it out of the box. Views expose a collection called **`GestureRecognizers`**, of type **`IList<GestureRecognizer>`**. Supported gesture recognizers are:

- **`TapGestureRecognizer`**: Allows recognition of taps.
- **`PinchGestureRecognizers`**: Allows recognition of the pinch-to-zoom gesture.
- **`PanGestureRecognizers`**: Enables the dragging of objects with the pan gesture.

For example, the following XAML demonstrates how to add a **`TapGestureRecognizer`** to an **`Image`** control:

```
<Image Source="https://www.xamarin.com/content/images/pages/branding/assets/x
amarin-logo.png" Aspect="AspectFit">
   <Image.GestureRecognizers>
       <TapGestureRecognizer x:Name="ImageTap"
        NumberOfTapsRequired="1" Tapped="ImageTap_Tapped"/>
   </Image.GestureRecognizers>
</Image>
```

You can assign the **NumberOfTapsRequired** property and the **Tapped** event with a handler that will be invoked when the user taps the image. It will look like this:

```
private void ImageTap_Tapped(object sender, EventArgs e)
{
    // Do your stuff here...
}
```

Gesture recognizers give you great flexibility and allow you to improve the user experience in your mobile apps by adding touch support where required.

## Displaying alerts

All platforms can show pop-up alerts with informative messages or receive user input with common choices such as OK or Cancel. Pages in Xamarin.Forms provide an asynchronous method called **DisplayAlert**, which is very easy to use. For example, suppose you want to display a message when the user taps a button. The following code demonstrates how to accomplish this:

```
private async void Button1_Clicked(object sender, EventArgs e)
{
    await DisplayAlert("Title", "This is an informational pop-up", "OK");
}
```

As an asynchronous method, you call **DisplayAlert** with the **await** operator, marking the containing method as **async**. The first argument is the pop-up title, the second argument is the text message, and the third argument is the text you want to display in the only button that appears. **DisplayAlert** has an overload that can wait for the user input and return **true** or **false** depending on whether the user selected the OK option or the Cancel option:

```
bool result =
     await DisplayAlert("Title", "Do you wish to continue?", "OK", "Cancel");
```

You are free to write whatever text you like for the OK and Cancel options, and IntelliSense helps you understand the order of these options in the parameter list. If the user selects OK, **DisplayAlert** returns **true**. Figure 41 shows an example of the alert.

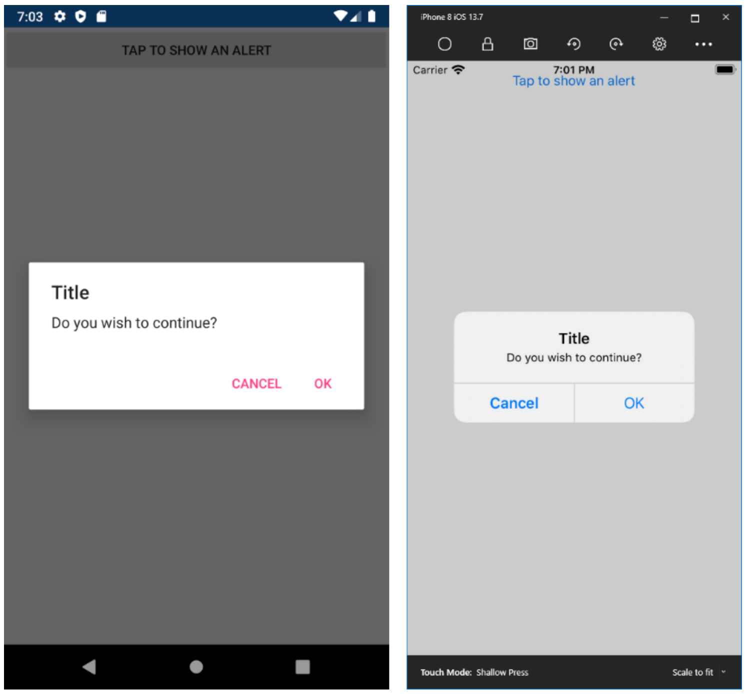


*Figure 41: Displaying alerts*

## Introducing the Visual State Manager

With the Visual State Manager, you can make changes to the user interface based on a view's state, such as **`Normal`**, **`Focused`**, and **`Disabled`**, all with declarative code. For example, you can use the Visual State Manager to change the color of a view depending on its state. The following code snippet demonstrates how you can change the background color of an **`Entry`** view based on its state:

```
<Entry>
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="White" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Focused">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor"
                            Value="LightGray" />
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="Disabled">
                <VisualState.Setters>
```

```
                        <Setter Property="BackgroundColor" Value="Gray" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
    </Entry>
```

With this markup code, the **`Entry`** will automatically change its background color when its state changes. In this case, you will need to set the **`Entry`**’s **`IsEnabled`** property to **`False`** in order to disable the view and trigger the **`Disabled`** state.

States must be grouped into objects called **`VisualStateGroup`**. Each state is represented by the **`VisualState`** object, where you add **`Setter`** specifications as you would do with styles, therefore providing the name of the property you want to change, and its value. Of course, you can specify multiple property setters.

Xamarin.Forms defines a set of states called common states, such as **`Normal`**, **`Focused`**, and **`Disabled`** (see the **`VisualStateGroup`** with the **`CommonState`** name in the preceeding code); these states are common to each view. Other states might be available only to specific views, and not to other views. The Visual State Manager provides an elegant and clean way to control the user interface behavior, all in your XAML code.

## Chapter summary

This chapter introduced the concepts of view and common views in Xamarin.Forms, the building blocks for any user interface in your mobile apps. You have seen how to obtain user input with the **`Button`**, **`Entry`**, **`Editor`**, and **`SearchBar`** controls; you have seen how to display information with the **`Label`** and how to use it in conjunction with other input views such as **`Slider`**, **`Stepper`**, and **`Switch`**. You have seen how the **`DatePicker`** and **`TimePicker`** views allow you to work with dates and time.

You have seen how to display images with the **`Image`** view and videos with the **`MediaElement`**; you have used the **`WebView`** to show HTML content; and you have seen how to inform the user of the progress of long-running operations with **`ActivityIndicator`** and **`ProgressBar`**. You have seen how to add gesture support to views that do not include it out of the box, how to display alerts for informational purposes, and how to allow user choices. Finally, you have seen how the Visual State Manager allows changing the user interface behavior based on a view’s state, all with declarative code.

Now you have all you need to build high-quality user interfaces with layouts and views, and you have seen how to use all these building blocks in a single page. However, most mobile apps are made of multiple pages. The next chapter explains all the available page types in Xamarin.Forms and the navigation infrastructure.

# Chapter 6 Pages and Navigation

In the previous chapters, we went over the basics of layouts and views, which are the fundamental building blocks of the user interface in mobile applications. However, I demonstrated how to use layouts and views within a single page, while real-world mobile apps are made of multiple pages. Android, iOS, and Windows provide a number of different pages that allow you to display content in several ways and to provide the best user experience possible based on the content you need to present.

Xamarin.Forms provides unified page models you can use from your single, shared C# codebase that work cross-platform. It also provides an easy-to-use navigation framework, which is the infrastructure you use to move between pages. In addition, Xamarin.Forms 4.0 has also introduced a simplified architecture in one place with the Shell. Pages, navigation, and the Shell are the last pieces of the user interface framework you need to know to build beautiful, native apps with Xamarin.Forms.

***Note: In order to follow the examples in the first part of this chapter, create a new Xamarin.Forms solution. The name is up to you. Every time a new page is discussed, just clean the content of the MainPage.xaml and MainPage.xaml.cs files (except for the constructor) and write the new code. When I cover the Shell, I will demonstrate a specific project template instead.***

## Introducing and creating pages

Xamarin.Forms provides many page objects that you can use to set up the user interfaces of your applications. Pages are root elements in the visual hierarchy, and each page allows you to add only one visual element, typically a root layout with other layouts and visual elements nested inside the root.

From a technical point of view, all the page objects in Xamarin.Forms derive from the abstract **`Page`** class, which provides the basic infrastructure of each page, including common properties such as **`Content`**. This is definitely the most important property that you assign with the root visual element. Table 6 describes available pages in Xamarin.Forms.

*Table 6: Pages in Xamarin.Forms*

| Page Type | Description |
|---|---|
| **`ContentPage`** | Displays a single view object. |
| **`TabbedPage`** | Facilitates navigating among child pages using tabs. |
| **`CarouselPage`** | Facilitates using the swipe gesture among child pages. |

| Page Type | Description |
|---|---|
| `FlyoutPage` | Manages two separate panes, and includes a flyout control. |
| `NavigationPage` | Provides the infrastructure for navigating among pages. |

The next sections describe the available pages in more detail. Remember that Visual Studio provides item templates for different page types, so you can right-click the .NET Standard project in Solution Explorer, select **Add New Item**, and in the **Add New Item** dialog box, you will see templates for each page described in Table 6.

***Note: If you read previous editions of this book, or if you have worked with Xamarin.Forms in the past, you will notice that in Table 6 I omitted mentioning the*** `MasterDetailPage` ***object. The reason is that it is now deprecated; you will want to use the*** `FlyoutPage` ***instead, or the Shell. Of course, for existing code, the*** `MasterDetailPage` ***is still certainly supported.***

## Single views with the ContentPage

The `ContentPage` object is the simplest page possible and allows for displaying a single visual element. You already looked at some examples of the `ContentPage` previously, but it is worth mentioning its `Title` property. This property is particularly useful when the `ContentPage` is used in pages with built-in navigation, such as `TabbedPage` and `CarouselPage`, because it helps identify the active page.

The core of the `ContentPage` is the `Content` property, which you assign with the visual element you want to display. The visual element can be either a single control or a layout; the latter allows you to create complex visual hierarchies and real-world user interfaces. In XAML, the tag for the `Content` property can be omitted, which is also common practice (also notice `Title`):

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <Label Text="A content page"/>

</ContentPage>
```

The `ContentPage` can be used individually, or as the content of other pages discussed in the next sections.

## Splitting contents with the FlyoutPage

The **`FlyoutPage`** is a very important page, since it allows you to split contents into two separate categories: generic and detailed. The user interface provided by the **`FlyoutPage`** is very common in Android and iOS apps. It offers a flyout on the left (the flyout part) that you can swipe to show and hide it, and a second area on the right that displays more detailed content (the detail part).

For example, a very common scenario for this kind of page is displaying a list of topics or settings in the flyout and the content for the selected topic or setting in the detail. Both the flyout and the detail parts are represented by **`ContentPage`** objects. A typical declaration for a **`FlyoutPage`** looks like Code Listing 12.

*Code Listing 12*

```
<?xml version="1.0" encoding="utf-8" ?>
<FlyoutPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <FlyoutPage.Flyout>
        <ContentPage Title="Main page">
            <Label Text="This is the Master" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </FlyoutPage.Flyout>
    <FlyoutPage.Detail>
        <ContentPage>
            <Label Text="This is the Details" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </FlyoutPage.Detail>
</MasterDetailPage>
```

As you can see, you populate the **`Flyout`** and **`Detail`** properties with the appropriate **`ContentPage`** objects. In real-world apps, you might have a list of topics in the **`Flyout`**, and then you might show details for a topic in the **`Detail`** when the user taps one in the **`Flyout`**’s content. Remember that assigning the **`Title`** property on the **`ContentPage`** that acts as the master is mandatory—otherwise, the runtime will throw an exception.

***Note: Every time you change the root page from*** *`ContentPage`* ***to another kind of page, such as*** *`FlyoutPage`****, you also need to change the inheritance in the code-behind. For example, if you open the C# MainPage.xaml.cs file, you will see that*** *`MainPage`* ***inherits from*** *`ContentPage`****, but in XAML you replaced this object with*** *`FlyoutPage`****. So, you also need to make*** *`MainPage`* ***inherit from*** *`FlyoutPage`****. If you***

***forget this, the compiler will report an error. This note is valid for the pages discussed in the next sections as well.***

Figures 42 and 43 show the flyout and detail parts, respectively. You can simply swipe from the left to enable the master flyout, and then swipe back to hide it. You can also control the flyout appearance programmatically by assigning the **`FlyoutLayoutBehavior`** with one of the following values:

- **`Default`**: The pages are displayed using each platform's default layout.
- **`Popover`**: The detail page covers the flyout page.
- **`Split`**: This equally splits the flyout page and the detail. The flyout is displayed on the left and the detail on the right.
- **`SplitOnLandscape`**: Similar to **`Split`**, but only applies when the device is in landscape orientation, otherwise **`Default`** is assumed.
- **`SplitOnPortrait`**: Similar to **`Split`**, but only applies when the device is in portrait orientation, otherwise **`Default`** is assumed.

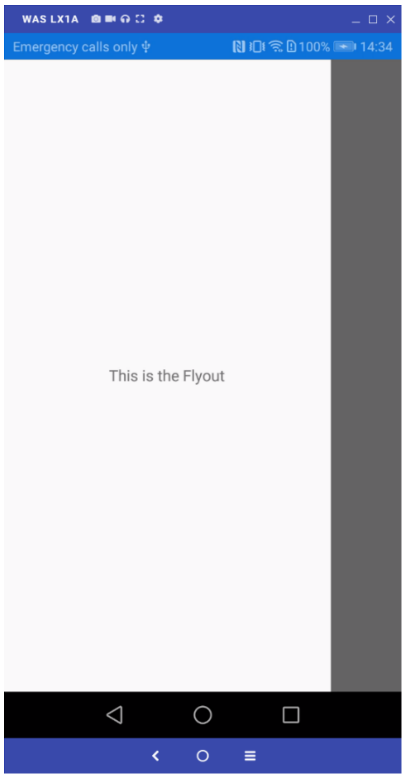


*Figure 42: FlyoutPage: The flyout*

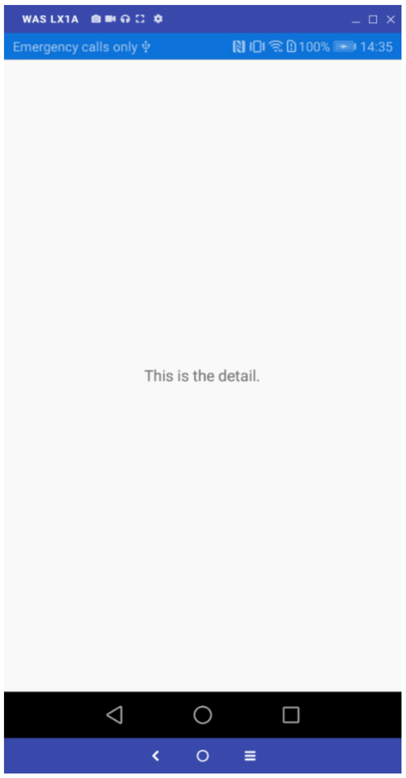


*Figure 43: FlyoutPage: The detail*

Another interesting property is called **`IsPresented`**, which you assign with **`true`** (visible) or **`false`** (hidden), and that is useful when the app is in landscape mode, because the flyout is automatically opened by default. When not specified, **`IsPresented`** is **`false`**.

## Displaying content within tabs with the TabbedPage

Sometimes you might need to categorize multiple pages by topic, or by activity type. When you have a small amount of content, you can take advantage of the **`TabbedPage`**, which can group multiple **`ContentPage`** objects into tabs for easy navigation. The **`TabbedPage`** can be declared as shown in Code Listing 13.

*Code Listing 13*

```
<?xml version="1.0" encoding="utf-8" ?>
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <TabbedPage.Children>
        <ContentPage Title="First">
            <Label Text="This is the first page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Second">
           <Label Text="This is the second page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Third">
            <Label Text="This is the third page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
    </TabbedPage.Children>
</TabbedPage>
```

As you can see, you populate the **`Children`** collection with multiple **`ContentPage`** objects. Providing a **`Title`** to each **`ContentPage`** is of primary importance, since the title’s text is displayed in each tab, as demonstrated in Figure 44.

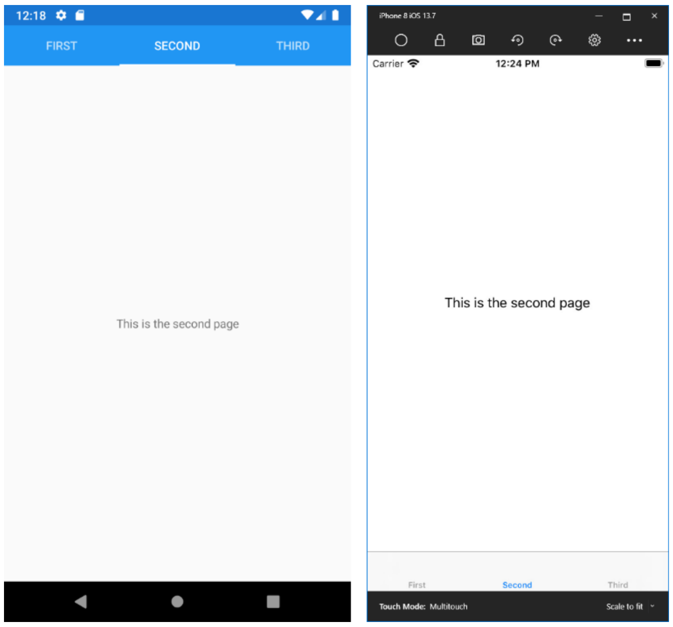


*Figure 44: Displaying grouped contents with the TabbedPage*

Of course, the **`TabbedPage`** works well with a small number of child pages, typically between three and four pages.

## Swiping pages with the CarouselPage

The **`CarouselPage`** is similar to the **`TabbedPage`**, but instead of having tabs, you can use the swipe gesture to switch among child pages. For example, the **`CarouselPage`** could be perfect to display a gallery of pictures. Code Listing 14 shows how to declare a **`CarouselPage`**.

*Code Listing 14*

```
<?xml version="1.0" encoding="utf-8" ?>
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <CarouselPage.Children>
        <ContentPage Title="First">
            <Label Text="This is the first page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Second">
           <Label Text="This is the second page" HorizontalOptions="Center"
                   VerticalOptions="Center"/>
        </ContentPage>
        <ContentPage Title="Third">
```

```
                <Label Text="This is the third page" HorizontalOptions="Center"
                       VerticalOptions="Center"/>
            </ContentPage>
        </CarouselPage.Children>
</CarouselPage>
```

Figure 45 shows how the **`CarouselPage`** appears.

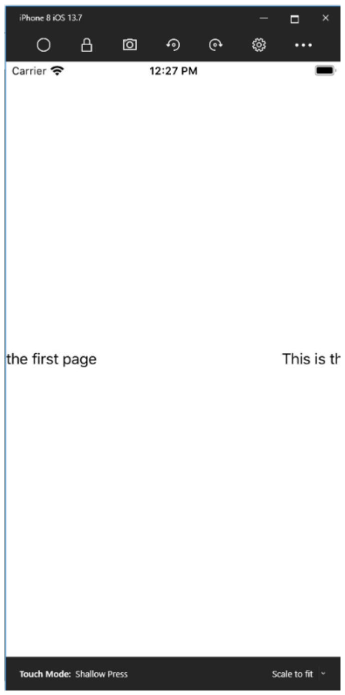


*Figure 45: Swiping contents with the CarouselPage*

## Navigating among pages

***Note: In the spirit of the Succinctly series, this section explains the most important concepts and topics of page navigation. However, there are tips and considerations that are specific to each platform that you have to know when dealing with navigation in Xamarin.Forms. To learn more about them, see the official documentation.***

Most mobile apps offer their content through multiple pages. In Xamarin.Forms, navigating among pages is very simple because of a built-in navigation framework. First of all, in Xamarin.Forms you leverage navigation features through the **`NavigationPage`** object. This kind of page must be instantiated, passing an instance of the first page in the stack of navigation to its constructor. This is typically done in the **App.xaml.cs** file, where you replace the assignment of the **`MainPage`** property with the following code:

```
public App()
```

```
{
    InitializeComponent();

    MainPage = new NavigationPage(new MainPage());
}
```

Wrapping a root page into a **NavigationPage** will not only enable the navigation stack, but will also enable the navigation bar on Android, iOS, and Windows desktop (but not on Windows 10 Mobile, which relies on the hardware Back button), whose text will be the value of the **Title** property of the current page object, represented by the **CurrentPage** read-only property. Now suppose you added another page of type **ContentPage** to the .NET Standard project, called **SecondaryPage.xaml**. The content of this page is not important at this point; just set its **Title** property with some text. If you want to navigate from the first page to the second page, use the **PushAsync** method as follows:

```
await Navigation.PushAsync(new SecondaryPage());
```

The **Navigation** property, exposed by each **Page** object, represents the navigation stack at the application level and provides methods for navigating between pages in a LIFO (last-in, first-out) approach. **PushAsync** navigates to the specified page instance; **PopAsync**, invoked from the current page, removes the current page from the stack and goes back to the previous page. Similarly, **PushModalAsync** and **PopModalAsync** allow you to navigate between pages modally. The following lines of code demonstrate this:

```
// removes SecondaryPage from the stack and goes back to the previous page
await Navigation.PopAsync();

// displays the specified page as a modal page
await Navigation.PushModalAsync(new SecondaryPage());
await Navigation.PopModalAsync();
```

Figure 46 shows how the navigation bar appears on Android and iOS when navigating to another page.

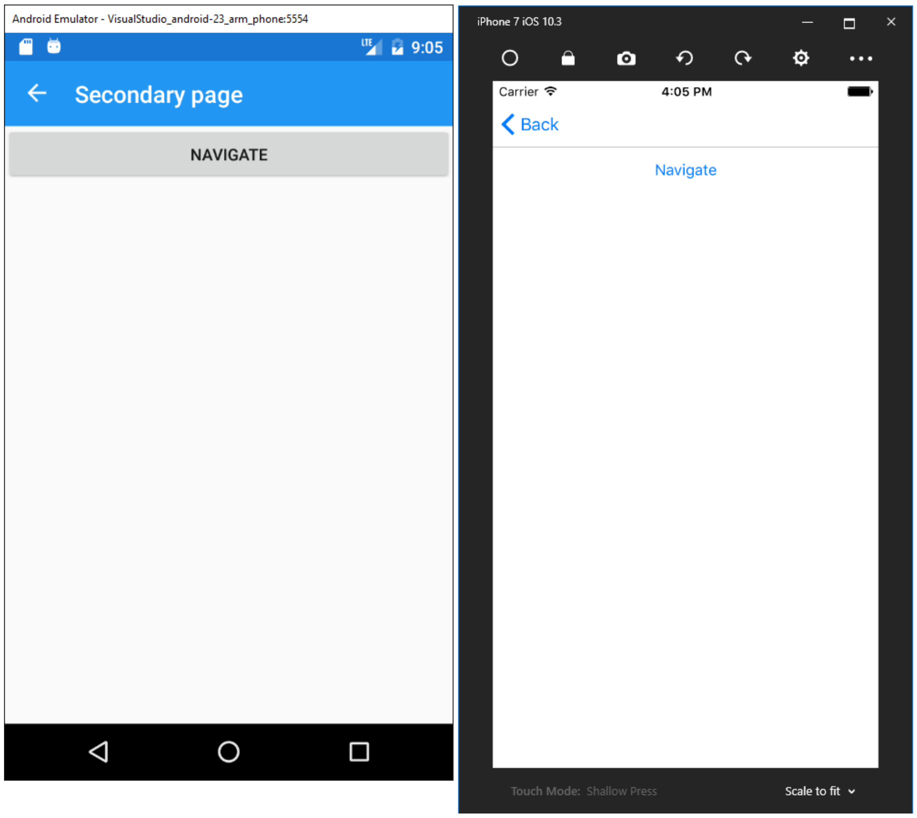

Android Emulator - VisualStudio_android-23_arm_phone:5554
9:05
Secondary page
NAVIGATE
iPhone 7 iOS 10.3
Carrier
4:05 PM
Back
Navigate
Touch Mode: Shallow Press
Scale to fit

*Figure 46: The navigation bar offered by the NavigationPage object*

Users can simply tap the Back button on the navigation bar to go back to the previous page. However, when you implement modal navigation, you cannot take advantage of the built-in navigation mechanism offered by the navigation bar, so it is your responsibility to implement code that allows the user to go back to the previous page.

Modal navigation can be useful if you must be able to intercept a tap on the Back button on each platform. In fact, Android and Windows devices have a built-in hardware Back button that you can manage with events, but iOS does not. In iOS, you only have the Back button provided by the navigation bar, but this cannot be accessed by any events. So, in this case, modal navigation can be a good option to intercept user actions.

## Passing objects between pages

The need to exchange data between pages is not uncommon. You can change or overload a `Page`'s constructor and require a parameter of the desired type. Then, when you call `PushAsync` and pass the instance of the new page, you will be able to supply the argument that is necessary to the new page's constructor.

## Animating transitions between pages

By default, the navigation includes an animation that makes the transition from one page to another nicer. However, you can disable animations by simply passing **`false`** as the argument of **`PushAsync`** and **`PushModalAsync`**.

## Managing the page lifecycle

Every **`Page`** object exposes the **`OnAppearing`** and **`OnDisappearing`** events, raised right before the page is rendered, and right before the page is removed from the stack, respectively. Their code looks like the following:

```
protected override void OnAppearing()
{
    // Replace with your code…
    base.OnAppearing();
}

protected override void OnDisappearing()
{
    // Replace with your code…
    base.OnDisappearing();
}
```

Actually, these events are not strictly related to navigation, since they are available to any page, including individual pages. However, it is with navigation that they become very important, especially when you need to execute some code at specific moments in the page lifecycle.

For a better understanding of the flow, think of the page constructor: this is invoked the very first time a page is created. Then, **`OnAppearing`** is raised right before the page is rendered on screen. When the app navigates to another page, **`OnDisappearing`** is invoked, but this does not destroy the current page instance (and this makes perfect sense). When the app navigates back from the second page to the first page, this is not created again because it is still in the navigation stack, so its constructor will not be invoked, while **`OnAppearing`** will. So, within the **`OnAppearing`** method body, you can write code that will be executed every time the page is shown, while in the constructor, you can write code that will be executed only once.

## Handling the hardware Back button

Android devices and Windows phones have a built-in hardware Back button that users can use instead of the Back button in the navigation bar. You can detect if the user presses the hardware Back button by handling the **`OnBackButtonPressed`** event as follows:

```
protected override bool OnBackButtonPressed()
{
    return base.OnBackButtonPressed(); // replace with your logic here
}
```

Simply put your logic in the method body. The default behavior is to suspend the app, so you might want to override this with `PopAsync` to return to the previous page. This event does not intercept pressing the Back button in the navigation bar, which implies it has no effect on iOS devices.

## Simplified app architecture: the Shell

More often than not, mobile apps share a number of features like a navigation bar, a search bar, and a flyout menu that users can open by sliding from the left side of the screen. Implementing these features by yourself is not complex at all, but it requires a certain amount of time. Moreover, it could be a repetitive task over multiple projects.

Starting with Xamarin.Forms 4.0, developers can leverage a new root layout called Shell. With the Shell, you can define the whole app architecture and hierarchy in one place, also providing a built-in navigation mechanism, search bar, and flyout menu.

In this chapter, you will learn the most common features of the Shell. However, this is a very sophisticated layout, so I suggest you bookmark the link to the official documentation page to keep as a reference for further studies.

***Note: At the time of this writing, the Shell is only available to Android and iOS apps. Support for UWP is currently under development. In fact, you will see that UWP as a target is disabled when creating Shell-based projects.***

Visual Studio 2019 offers a specific project template that implements the Shell, called **Flyout** (see Figure 4). Because the code generated by this project template is a bit complex if you are new to the Shell, I will not use such a project template. I will, instead, show how to create an app with basic functionalities based on the sample project called **ShellDemo**, located in the **Chapter6** folder of the companion repository. As you will see, the Shell definition resides in the **AppShell.xaml** file, which is the place where all the code discussed shortly needs to be placed.

***Tip: Once you get started with the Shell, you might want to have a look at an open-source example provided Microsoft, called Xaminals. This is a more advanced project that demonstrates how to implement more complex navigation, tabbed pages, and much more.***

### Structure of the Shell

At its core, the Shell is a container of the following visual elements:

- **Flyout menu**: This is a side menu that can be shown or hidden with a swipe gesture, and that can contain other visual elements or shortcuts to other pages. This is what Android users call the “hamburger” menu.
- **Tab bar**: This is usually placed at the bottom of the Shell and includes buttons with icons and text that users can tap to navigate to different pages.
- **Search bar**: This is used to search items in a list, typically by filtering a data-bound collection.

Visual elements in the Shell, such as flyout items and bar buttons, can be styled through resources. The flyout menu, tab bar, and search bar are independent from one another, and you can implement just one of them. In order to implement a flyout, you just need to add as many **FlyoutItem** objects to the Shell as you want.

If you only want a tab bar instead of the flyout menu, you can add a **TabBar** object to the Shell. Then to the **TabBar** you will add as many **Tab** objects, as many buttons, as you want to implement for navigation. Then, you will assign the **FlyoutBehavior** property of the Shell with **Disabled**. Both **FlyoutItem** and **Tab** need to be populated with an object of type **ShellContent** that points to the page you want to open.

If you want to have both the flyout and the tab bar, you will nest a **Tab** object into a **FlyoutItem** object. If this all seems confusing, do not worry; you will get proper examples in the next paragraphs.

## Implementing the flyout menu

Suppose you want to create a flyout menu with three items, each pointing to a specific page. The code looks like the following:

```
<Shell … >
    <FlyoutItem Title="Home" Icon="home.png">
        <ShellContent ContentTemplate="{DataTemplate pages:HomePage}"/>
    </FlyoutItem>
    <FlyoutItem Title="About" Icon="about.png">
        <ShellContent ContentTemplate="{DataTemplate pages:AboutPage}"/>
    </FlyoutItem>
    <FlyoutItem Title="Contact" Icon="contact.png">
        <ShellContent ContentTemplate="{DataTemplate pages:ContactPage}"/>
    </FlyoutItem>
</Shell>
```

For each **FlyoutItem**, you can specify the **Title**, which is the text you see in the menu, and an icon (icon files must follow the rules of each operating system). The **ShellContent** object represents the target page of the navigation.

The syntax based on the **ContentTemplate**, which receives a **DataTemplate** and the name of the target page as an argument, makes sure the instance of the target page is created only when required. You could also use the **Content** property instead of **ContentTemplate**, passing the name of the page directly, but in this case each page would be instantiated directly with potential performance overhead, so this is not recommended.

Data templates will be discussed in more details in the next chapter as part of the data-binding topic. The result of this code is shown in Figure 47.

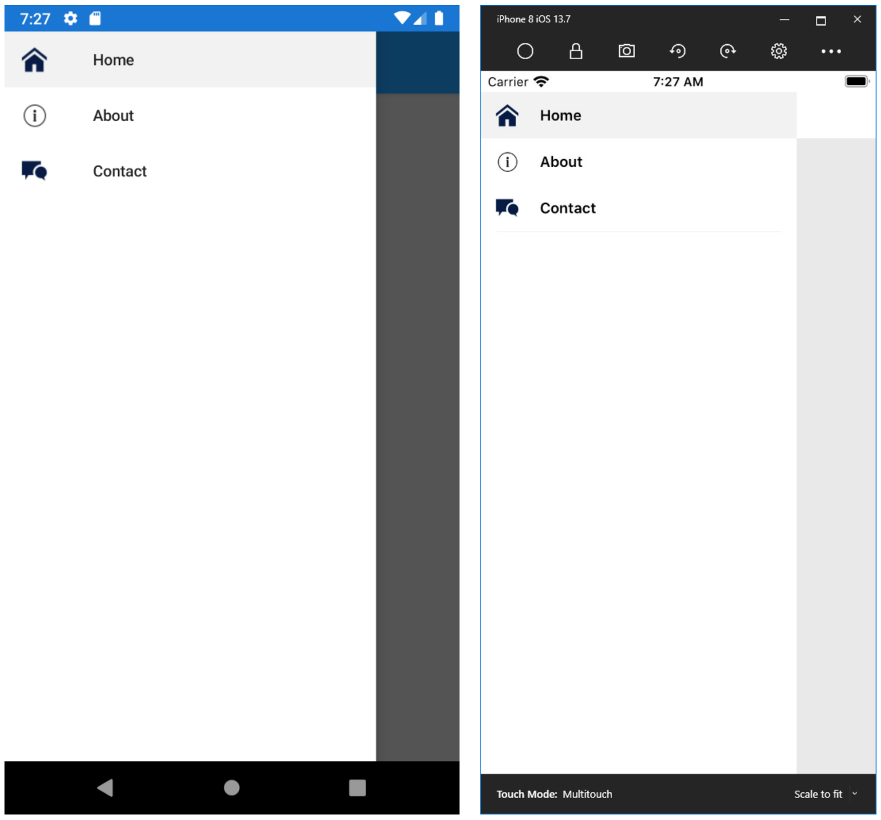


*Figure 47: Flyout menu with the Shell*

The flyout menu can be further customized via the following properties:

- **`FlyoutIcon`**: Sets the icon for the flyout.
- **`FlyoutHeader`**: Assigned with a view that appears at the top of the flyout. You can alternatively use the **`FlyoutHeaderTemplate`** property, which is populated with a **`DataTemplate`**.
- **`FlyoutBackgroundImage`**: Sets a background image for the flyout.
- **`FlyoutBackgroundImageAspect`**: Sets the aspect for the background image with the same values used for regular images (**`Fill`**, **`AspectFill`**, **`AspectFit`**).
- **`FlyoutIsPresented`**: Gets or sets the appearance status of the flyout. It can be also used in C# code to programmatically control the flyout appearance.

For a full list and description of the flyout properties, refer to the [documentation page](documentation page).

## Implementing the tab bar

Implementing the tab bar is accomplished by adding a **`TabBar`** object to the Shell, and then adding as many **`Tab`** items for as many navigation buttons you want. Continuing the previous example of three items, the code would look like this:

```
<Shell … >
    <TabBar>
        <Tab Title="Home" Icon="home.png">
```

```
            <ShellContent ContentTemplate="{DataTemplate pages:HomePage}"/>
        </Tab>
        <Tab Title="About" Icon="about.png">
            <ShellContent ContentTemplate="{DataTemplate pages:AboutPage}"/>
        </Tab>
        <Tab Title="Contact" Icon="contact.png">
            <ShellContent
                        ContentTemplate="{DataTemplate pages:ContactPage}"/>
        </Tab>
    </TabBar>
</Shell>
```

As you can see, each **`Tab`** has a title and an icon like the **`FlyoutItem`**. Navigation is again performed via the **`ShellContent`** object. Notice how you do not need to handle any events to perform navigation—everything is handled by the Shell. Figure 48 shows how the tab bar looks.

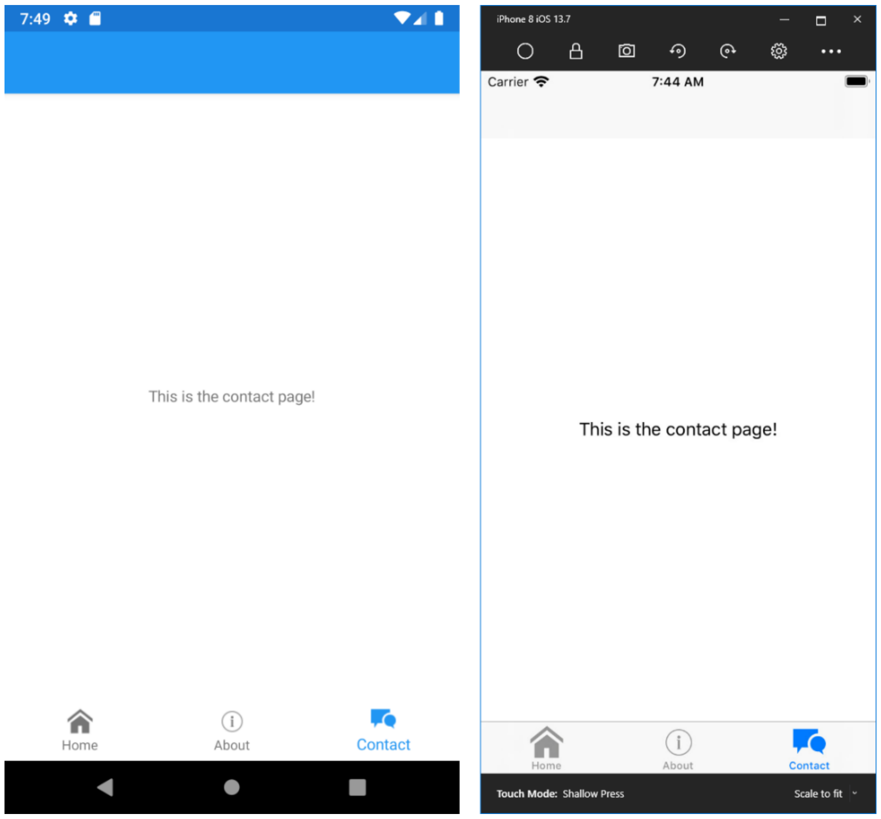


*Figure 48: Tab bar with the Shell*

***Note: The Shell takes care of navigation between pages with its built-in mechanism, but you have deep control of it, and you can even manage navigation at***

***runtime. This is known as routing, and I recommend you have a look at the documentation for more information.***

## Implementing the flyout with tab bar

If you want to have the same elements in both the flyout and the tab bar, you can declare one **FlyoutItem** and add **Tab** objects inside it, like in the following code:

```
<Shell … >
    <FlyoutItem FlyoutDisplayOptions="AsMultipleItems">
        <Tab Title="Home" Icon="home.png">
            <ShellContent ContentTemplate="{DataTemplate pages:HomePage}"/>
        </Tab>
        <Tab Title="About" Icon="library.png">
            <ShellContent ContentTemplate="{DataTemplate pages:AboutPage}"/>
        </Tab>
        <Tab Title="Contact" Icon="contact.png">
            <ShellContent
                        ContentTemplate="{DataTemplate pages:ContactPage}"/>
        </Tab>
    </FlyoutItem>
</Shell>
```

In this case, you set the **FlyoutDisplayOptions** property with **AsMultipleItems**, which allows for displaying several elements in a single flyout item. You can run the code and see how the app will show both the flyout with its items and the navigation bar.

## Adding the search bar

Another piece of the Shell is the integrated search tool. The search tool is made of a class that derives from **SearchHandler**. Code Listing 15 shows the structure of this class.

*Code Listing 15*

```
    public class ItemsSearchHandler : SearchHandler
    {
        protected override void OnQueryChanged(string oldValue,
                  string newValue)
        {
            base.OnQueryChanged(oldValue, newValue);

            if (string.IsNullOrWhiteSpace(newValue))
            {
                ItemsSource = null;
            }
            else
            {
                ItemsSource = YourCollection
```

```
                    .Where(p =>
p.YourProperty.ToLower().Contains(newValue.ToLower()))
                    .ToList();
            }
        }

        protected override async void OnItemSelected(object item)
        {
            base.OnItemSelected(item);

            await (App.Current.MainPage as Shell).
                   GoToAsync($"ObjectDetails?name={((YourObject)item).
                   YourProperty}");
        }
    }
```

In the **OnQueryChanged** event handler, which is invoked when the user types in the search box, you will filter the collection bound to your page. In Code Listing 15, **YourCollection** is a collection that implements the **IEnumerable<T>** interface, and that can be assigned to the **ItemsSource** property as the data source for the search tool, whereas **YourProperty** is a property on which the filter is based.

The **oldValue** and **newValue** string arguments of **OnQueryChanged** allow you to get the value in the search box before and after typing into it. The **OnItemSelected** event is raised when the user selects one of the items displayed in the search tool. It allows for navigating to a specific page. The example assumes there is a page called **ObjectDetails** to which the instance of the selected object is passed via query string as an argument, after conversion.

From the XAML point of view, you would add the following code inside the **Shell** definition:

```
<Shell.SearchHandler>

    <local:ItemsSearchHandler Placeholder="Enter search term"

                              ShowsResults="true"

                              DisplayMemberName="YourProperty" />

</Shell.SearchHandler>
```

The **SearchHandler** property is assigned with the **SearchHandler** implementation. It allows for specifying a placeholder text if the search box must show a preview of the results, and the name of the property from the bound collection that will be displayed (**DisplayMemberName**). Figure 49 shows an example based on the Xaminals sample project from Microsoft.

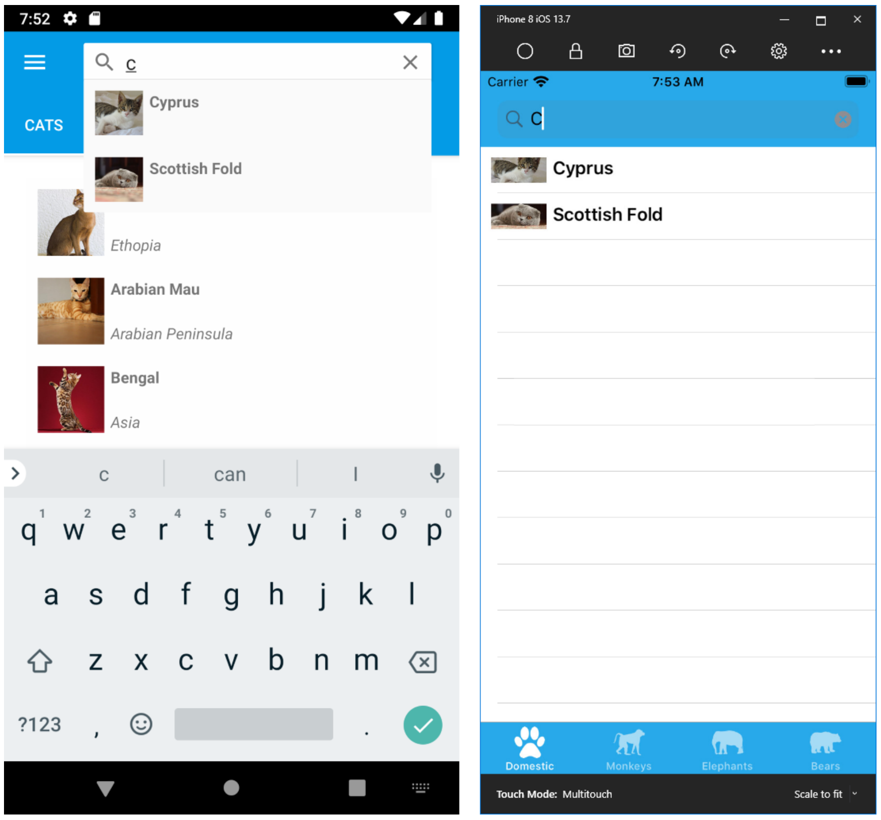


*Figure 49: The search tool within the Shell*

## Styling the Shell

Elements in the Shell, such as the flyout and tab bar, can be customized with different colors and fonts. This is accomplished by defining specific resources. Actually, resources in Xamarin.Forms are discussed in the next chapter, so here you get a preview. For example, you could apply custom colors to the Shell and to tab bar elements with the following code:

```
<Shell.Resources>
    <ResourceDictionary>
        <Style x:Key="BaseStyle" TargetType="Element">
            <Setter Property="Shell.BackgroundColor" Value="LightGreen" />
            <Setter Property="Shell.ForegroundColor" Value="White" />
            <Setter Property="Shell.TitleColor" Value="White" />
            <Setter Property="Shell.DisabledColor" Value="#B4FFFFFF" />
            <Setter Property="Shell.UnselectedColor" Value="#95FFFFFF" />
            <Setter Property="Shell.TabBarBackgroundColor"
                    Value="LightBlue"/>
            <Setter Property="Shell.TabBarForegroundColor" Value="White"/>
            <Setter Property="Shell.TabBarUnselectedColor"
                    Value="#95FFFFFF"/>
            <Setter Property="Shell.TabBarTitleColor" Value="White"/>
       </Style>
  </ResourceDictionary>
```

```
</Shell.Resources>
```

Names for properties that accept styling are really self-explanatory. You can also customize the appearance of the flyout and its elements. There are several additional ways to customize and configure the Shell. I recommend that you read the documentation for further studies.

***Note: There is much more to say about the Shell, which is a very sophisticated tool. This book could not cover topics such as custom navigation settings, custom renderers, and programmatic access to the Shell. If you consider implementing the Shell in your apps, I strongly recommend you bookmark the root of the official documentation, which contains everything you need to create full Shell experiences.***

## Chapter summary

This chapter introduced the available pages in Xamarin.Forms, explaining how you can display single-view content with the **`ContentPage`** object, group content into tabs with the **`TabbedPage`**, swipe content with the **`CarouselPage`**, and group contents into two categories with the **`MasterDetail`** page object.

You also looked at how the **`NavigationPage`** object provides a built-in navigation framework that not only displays a navigation bar, but also allows for navigating between pages programmatically.

Finally, you got an introduction to the Shell, an object that simplifies the app infrastructure by including a flyout menu, a tab bar, and a search box in one place.

In the next chapter, you will look at information about two important and powerful features in Xamarin.Forms: resources and data binding.

# Chapter 7  Resources and Data Binding

XAML is a very powerful declarative language, and it shows all of its power with two particular scenarios: working with resources and working with data binding. If you have existing experience with platforms like WPF, Silverlight, and Universal Windows Platform, you will be familiar with the concepts described in this chapter. If this is your first time, you will immediately appreciate how XAML simplifies difficult things in both scenarios.

## Working with resources

Generally speaking, in XAML-based platforms such as WPF, Silverlight, Universal Windows Platform, and Xamarin.Forms, resources are reusable pieces of information that you can apply to visual elements in the user interface. Typical XAML resources are styles, control templates, object references, and data templates. Xamarin.Forms supports styles and data templates, so these will be discussed in this chapter.

***Tip: Resources in XAML are very different from resources in platforms such as Windows Forms, where you typically use .resx files to embed strings, images, icons, or files. My suggestion is that you should not make any comparison between XAML resources and other .NET resources.***

### Declaring resources

Every **Page** object and layout exposes a property called **Resources**, a collection of XAML resources that you can populate with one or more objects of type **ResourceDictionary**. A **ResourceDictionary** is a container of XAML resources such as styles, data templates, and object references. For example, you can add a **ResourceDictionary** to a page as follows:

```
<ContentPage.Resources>
   <ResourceDictionary>
       <!-- Add resources here -->
   </ResourceDictionary>
</ContentPage.Resources>
```

Resources have scope. This implies that resources you add to the page level are available to the whole page, whereas resources you add to the layout level are only available to the current layout, like in the following snippet:

```
<StackLayout.Resources>
   <ResourceDictionary>
       <!-- Resources are available only to this layout, not outside -->
   </ResourceDictionary>
</StackLayout.Resources>
```

Sometimes you might want to make resources available to the entire application. In this case, you can take advantage of the **App.xaml** file. The default code for this file is shown in Code Listing 16.

*Code Listing 16*

```
<?xml version="1.0" encoding="utf-8" ?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="App1.App">
      <Application.Resources>

          <!-- Application resource dictionary -->
            <ResourceDictionary>

            </ResourceDictionary>
      </Application.Resources>
</Application>
```

As you can see, the autogenerated code of this file already contains an **`Application.Resources`** node with a nested **`ResourceDictionary`**. Resources you put inside this resource dictionary will be visible to any page, layout, and view in the application. Now that you have knowledge of where resources are declared and their scope, it is time to see how resources work, starting with styles. Other resources, such as data templates, will be discussed later in this chapter.

## Introducing styles

When designing your user interface, in some situations, you might have multiple views of the same type and, for each of them, you might need to assign the same properties with the same values. For example, you might have two buttons with the same width and height, or two or more labels with the same width, height, and font settings. In such situations, instead of assigning the same properties many times, you can take advantage of styles.

A style allows you to assign a set of properties to views of the same type. Styles must be defined inside a **`ResourceDictionary`**, and they must specify the type they are intended for and an identifier. The following code demonstrates how to define a style for **`Label`** views:

```
<ResourceDictionary>
   <Style x:Key="labelStyle" TargetType="Label">
      <Setter Property="TextColor" Value="Green" />
      <Setter Property="FontSize" Value="Large" />
   </Style>
</ResourceDictionary>
```

You assign an identifier with the **`x:Key`** expression and the target type with **`TargetType`**, passing the type name for the target view. Property values are assigned with **`Setter`** elements, whose **`Property`** property represents the target property name, and whose **`Value`** represents the property value. You then assign the style to **`Label`** views as follows:

```
<Label Text="Enter some text:" Style="{StaticResource labelStyle}"/>
```

A style is therefore applied by assigning the **`Style`** property on a view with an expression that encloses the **`StaticResource`** markup extension and the style identifier within curly braces. You can then assign the **`Style`** property on each view of that type instead of manually assigning the same properties every time. With styles, XAML supports both **`StaticResource`** and **`DynamicResource`** markup extensions. In the first case, if a style changes, the target view will not be updated with the refreshed style. In the second case, the view will be updated reflecting changes in the style.

## Style inheritance

Styles support inheritance; therefore, you can create a style that derives from another style. For example, you can define a style that targets the abstract **`View`** type as follows:

```
<Style x:Key="viewStyle" TargetType="View">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="Center" />
</Style>
```

This style can be applied to any view, regardless of the concrete type. Then you can create a more specialized style using the **`BasedOn`** property as follows:

```
<Style x:Key="labelStyle" TargetType="Label"
       BasedOn="{StaticResource viewStyle}">
    <Setter Property="TextColor" Value="Green" />
</Style>
```

The second style targets **`Label`** views, but also inherits property settings from the parent style. Put succinctly, the **`labelStyle`** will assign the **`HorizontalOptions`**, **`VerticalOptions`**, and **`TextColor`** properties on the targeted **`Label`** views.

## Implicit styling

A view's **`Style`** property allows the assigning of a style defined inside resources. This allows you to selectively assign the style only to certain views of a given type. However, if you want the same style to be applied to all of the views of the same type in the user interface, assigning the **`Style`** property to each view manually might be tedious. In this case, you can take advantage of *implicit styling*. This feature allows you to assign a style automatically to all the views of the type specified with the **`TargetType`** property without the need to set the **`Style`** property. To accomplish this, you simply avoid assigning an identifier with **`x:Key`**, like in the following example:

```
<Style TargetType="Label">
   <Setter Property="HorizontalOptions" Value="Center" />
   <Setter Property="VerticalOptions" Value="Center" />
   <Setter Property="TextColor" Value="Green" />
</Style>
```

Styles with no identifier will automatically be applied to all the **`Label`** views in the user interface (according to the scope of the containing resource dictionary), and you will not need to assign the **`Style`** property on the **`Label`** definitions.

## Working with data binding

Data binding is a built-in mechanism that allows visual elements to communicate with data so that the user interface is automatically updated when data changes, and vice versa. Data binding is available in all the most important development platforms, and Xamarin.Forms is no exception. In fact, its data-binding engine relies on the power of XAML, and the way it works is similar in all the XAML-based platforms. Xamarin.Forms supports binding an object to visual elements, a collection to visual elements, and visual elements to other visual elements. This chapter describes the first two scenarios.

Because data binding is a very complex topic, the best way to start is with an example. Suppose you want to bind an instance of the following **`Person`** class to the user interface, so that a communication flow is established between the object and views:

```
public class Person
{
    public string FullName { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string Address { get; set; }
}
```

In the user interface, you will want to allow the user to enter their full name, date of birth, and address via an **`Entry`**, a **`DatePicker`**, and another **`Entry`**, respectively. In XAML, this can be accomplished with the code shown in Code Listing 17.

*Code Listing 17*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="20">
        <Label Text="Name:" />
        <Entry Text="{Binding FullName}"/>

        <Label Text="Date of birth:"/>
        <DatePicker Date="{Binding DateOfBirth, Mode=TwoWay}"/>

        <Label Text="Address:"/>
        <Entry Text="{Binding Address}"/>
    </StackLayout>
```

```
</ContentPage>
```

As you can see, the **`Text`** property for **`Entry`** views and the **`Date`** property of the **`DatePicker`** have a markup expression as their value. Such an expression is made up of the **`Binding`** literal followed by the property you want to bind from the data object. Actually, the expanded form of this syntax could be **`{Binding Path=PropertyName}`**, but **`Path`** can be omitted. Data binding can be of five types:

- **`TwoWay`**: Views can read and write data.
- **`OneWay`**: Views can only read data.
- **`OneWayToSource`**: Views can only write data.
- **`OneTime`**: Views can read data only once.
- **`Default`**: Xamarin.Forms resolves the appropriate mode automatically, based on the view (see the explanation that follows).

**`TwoWay`** and **`OneWay`** are the most-used modes, and in most cases, you do not need to specify the mode explicitly because Xamarin.Forms automatically resolves the appropriate mode based on the view. For example, binding in the **`Entry`** control is **`TwoWay`** because this kind of view can be used to read and write data, whereas binding in the **`Label`** control is **`OneWay`** because this view can only read data. However, with the **`DatePicker`**, you need to explicitly set the binding mode, so you use the following syntax:

```
<DatePicker Date="{Binding DateOfBirth, Mode=TwoWay}"/>
```

Views' properties that are bound to an object's properties are known as *bindable properties* (or dependency properties if you come from the WPF or UWP world).

***Tip: Bindable properties are very powerful, but a bit more complex in the architecture. In this chapter, I'm going to explain how to use them, but for further details about their implementation and how you can use them in your custom objects, you can refer to the official documentation.***

Bindable properties will automatically update the value of the bound object's property and will automatically refresh their value in the user interface if the object is updated. However, this automatic refresh is possible only if the data-bound object implements the **`INotifyPropertyChanged`** interface, which allows an object to send change notifications. As a consequence, you must extend the **`Person`** class definition, as shown in Code Listing 18.

*Code Listing 18*

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;

namespace App1
{
    public class Person : INotifyPropertyChanged
    {
        private string fullName;
        public string FullName
        {
            get
            {
                return fullName;
            }
            set
            {
                fullName = value;
                OnPropertyChanged();
            }
        }

        private DateTime dateOfBirth;

        public DateTime DateOfBirth
        {
            get
            {
                return dateOfBirth;
            }
            set
            {
                dateOfBirth = value;
                OnPropertyChanged();
            }
        }
        private string address;
        public string Address
        {
            get
            {
                return address;
            }
            set
            {
                address = value;
                OnPropertyChanged();
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        private void OnPropertyChanged([CallerMemberName]
                string propertyName = null)
        {
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

By implementing **`INotifyPropertyChanged`**, property setters can raise a change notification via the **`PropertyChanged`** event. Bound views will be notified of any changes and will refresh their contents.

***Tip: With the** `CallerMemberName` **attribute, the compiler automatically resolves the name of the caller member. This avoids the need to pass the property name in each setter and helps keep code much cleaner.***

The next step is binding an instance of the **`Person`** class to the user interface. This can be accomplished with the following lines of code, normally placed inside the page's constructor or in its **`OnAppearing`** event handler:

```
Person person = new Person();
this.BindingContext = person;
```

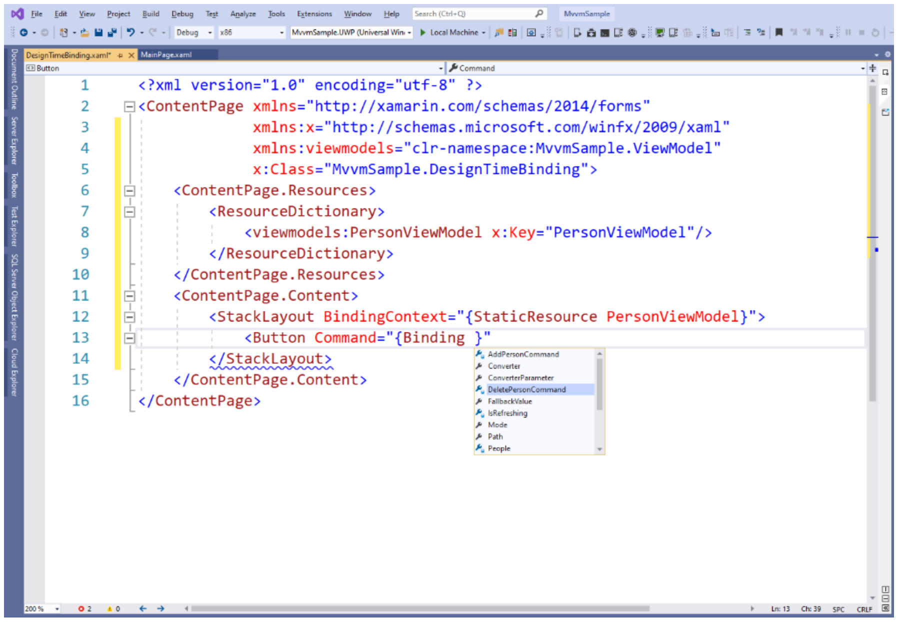
Pages and layouts expose the **`BindingContext`** property, of type **`object`**, which represents the data source for the page or layout and is the same as **`DataContext`** in WPF or UWP. Child views that are data bound to an object's properties will search for an instance of the object in the **`BindingContext`** property value and bind to properties from this instance. In this case, the **`Entry`** and the **`DatePicker`** will search for an object instance inside **`BindingContext`**, and they will bind to properties from that instance. Remember that XAML is case-sensitive, so binding to **`FullName`** is different from binding to **`Fullname`**. The runtime will throw an exception if you try to bind to a property that does not exist or has a different name.

If you now try to run the application, not only will data binding work, but the user interface will also be automatically updated if the data source changes. You may think of binding views to a single object instance, like in the previous example, as binding to a row in a database table.

## IntelliSense support for data binding and resources

IntelliSense provides full support for binding expressions with markup extensions. For instance, suppose you have a **`Person`** class you want to use as the binding context for your UI, and it is declared as a local resource. IntelliSense will help you create the binding expression by showing the list of available resources. With a binding context declared in the resources, IntelliSense can help with creating binding expressions by showing a list of properties exposed by the bound object.

Figure 50 shows an example where you can also see the **`Command`** property of a button being bound to the property **`DeletePersonCommand`**, which is defined in the view model.



```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:viewmodels="clr-namespace:MvvmSample.ViewModel"
             x:Class="MvvmSample.DesignTimeBinding">
    <ContentPage.Resources>
        <ResourceDictionary>
            <viewmodels:PersonViewModel x:Key="PersonViewModel"/>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout BindingContext="{StaticResource PersonViewModel}">
            <Button Command="{Binding }"
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

*Figure 50: IntelliSense support for data-binding expressions*

This is a big productivity feature that simplifies the way you create binding expressions.

## Bindable spans

Xamarin.Forms offers the so-called *bindable spans*. The **`Span`** class inherits from **`BindableObject`**, which means that all of its properties support data binding. The following snippet provides an example:

```
<Label
   HorizontalOptions="Center"
   VerticalOptions="CenterAndExpand">
    <Label.FormattedText>
        <FormattedString>
            <FormattedString.Spans>
                <Span FontSize="{Binding TitleSize}"
                      ForegroundColor="{Binding TitleColor}"
                      Text="{Binding Title}" />
                <Span FontSize="{Binding SubTitleSize}"
                      ForegroundColor="{Binding SubTitleColor}"
                      Text="{Binding SubTitle}" />
                <Span FontSize="{Binding AuthorSize}"
                      ForegroundColor="{Binding AuthorColor}"
                      Text="{Binding AuthorName}" />
            </FormattedString.Spans>
        </FormattedString>
    </Label.FormattedText>
</Label>
```

With bindable spans, you can create formatted strings dynamically, based on the data exposed by your view models.

## Working with collections

Though working with a single object instance is a common scenario, another very common situation is working with collections that you display as lists in the user interface. Xamarin.Forms supports data binding over collections via the **`ObservableCollection<T>`** object. This collection works exactly like the **`List<T>`**, but it also raises a change notification when items are added to or removed from the collection. Collections are very useful, for example, when you want to represent rows in a database table. For example, suppose you have the following collection of **`Person`** objects:

```
Person person1 = new Person { FullName = "Alessandro" };
Person person2 = new Person { FullName = "James" };
Person person3 = new Person { FullName = "Jacqueline" };
var people = new ObservableCollection<Person>() { person1, person2,
             person3 };

this.BindingContext = people;
```

The code assigns the collection to the **`BindingContext`** property of the root container, but at this point, you need a visual element that is capable of displaying the content of this collection. This is where the **`ListView`** control comes in. The **`ListView`** can receive the data source from either the **`BindingContext`** of its container or by assigning its **`ItemsSource`** property, and any object that implements the **`IEnumerable`** interface can be used with the **`ListView`**. You will typically assign **`ItemsSource`** directly if the data source for the **`ListView`** is not the same data source as for the other views in the page.

The problem to solve with the **`ListView`** is that it does not know how to present objects in a list. For example, think of the **`People`** collection that contains instances of the **`Person`** class. Each instance exposes the **`FullName`**, **`DateOfBirth`**, and **`Address`** properties, but the **`ListView`** does not know how to present these properties, so it is your job to explain to it how. This is accomplished with the *data template*, which is a static set of views that are bound to properties in the object. It instructs the **`ListView`** on how to present items. Data templates in Xamarin.Forms rely on the concept of cells. Cells can display information in a specific way and are summarized in Table 7.

*Table 7: Cells in Xamarin.Forms*

| Cell Type | Description |
|---|---|
| **`TextCell`** | Displays two labels: one with a description, and one with a data-bound text value. |
| **`EntryCell`** | Displays a label with a description and an **`Entry`** with a data-bound text value. It also allows a placeholder to be displayed. |

| Cell Type | Description |
|---|---|
| **`ImageCell`** | Displays a label with a description and an **`Image`** control with a data-bound image. |
| **`SwitchCell`** | Displays a label with a description and a **`Switch`** control bound to a **`bool`** value. |
| **`ViewCell`** | Allows for creating custom data templates. |

***Tip: Labels within cells are also bindable properties.***

For example, if you only had to display and edit the **`FullName`** property, you could write the following data template:

```
<Grid>
    <ListView x:Name="PeopleList" ItemsSource="{Binding}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <EntryCell Label="Full name:" Text="{Binding FullName}"/>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</Grid>
```

***Tip: The `DataTemplate` definition is always defined inside the `ListView.ItemTemplate` element.***

As a general rule, if the data source is assigned to the **`BindingContext`** property, the **`ItemsSource`** must be set with the **`{Binding}`** value, which means your data source is the same as that of your parent. With this code, the **`ListView`** will display all the items in the bound collection, showing two cells for each item. However, each **`Person`** also exposes a property of type **`DateTime`**, and no cell is suitable for that. In such situations, you can create a custom cell using the **`ViewCell`**, as shown in Code Listing 19.

*Code Listing 19*

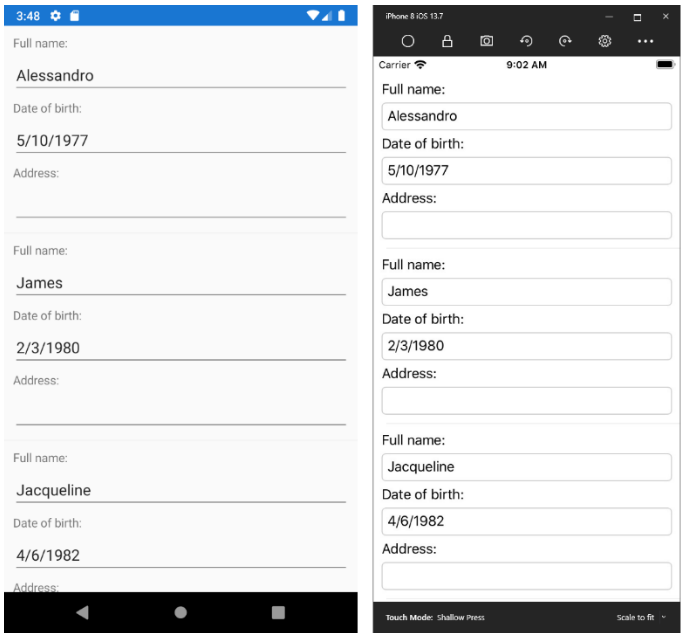
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page" Padding="20"
             x:Class="App1.MainPage">

    <StackLayout>
        <ListView x:Name="PeopleList" ItemsSource="{Binding}"
                  HasUnevenRows="True">
```

```
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <ViewCell.View>
                            <StackLayout Margin="10">
                                <Label Text="Full name:"/>
                                <Entry Text="{Binding FullName}"/>
                                <Label Text="Date of birth:"/>
                                <DatePicker Date="{Binding DateOfBirth,
                                            Mode=TwoWay}"/>
                                <Label Text="Address:"/>
                                <Entry Text="{Binding Address}"/>
                            </StackLayout>
                        </ViewCell.View>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

As you can see, the **`ViewCell`** allows you to create custom and complex data templates, contained in the **`ViewCell.View`** property, so that you can display whatever kind of information you need. Notice the **`HasUnevenRows`** property: if **`true`** on Android and Windows, this dynamically resizes a cell's height based on its content. On iOS, this property must be set to **`false`** and you must provide a fixed row height by setting the **`RowHeight`** property. In Chapter 9 you will learn how to take advantage of the **`OnPlatform`** object to make UI decisions based on the platform.

***Tip: The `ListView` is a very powerful and versatile view, and there is much more to it, such as interactivity, grouping and sorting, and customizations. I strongly recommend you read the official documentation and this article that describes how to improve performance, which is extremely useful with Android.***

Figure 51 shows the result for the code described in this section. Notice that the **`ListView`** includes built-in scrolling capability and must never be enclosed within a **`ScrollView`**.



Full name:
Alessandro
Date of birth:
5/10/1977
Address:

Full name:
James
Date of birth:
2/3/1980
Address:

Full name:
Jacqueline
Date of birth:
4/6/1982
Address:

Full name:
Alessandro
Date of birth:
5/10/1977
Address:

Full name:
James
Date of birth:
2/3/1980
Address:

Full name:
Jacqueline
Date of birth:
4/6/1982
Address:

*Figure 51: A data-bound ListView*

A data template can be placed inside the page or app resources so that it becomes reusable. Then you assign the **`ItemTemplate`** property in the **`ListView`** definition with the **`StaticResource`** expression, as shown in Code Listing 20.

*Code Listing 20*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <DataTemplate x:Key="MyTemplate">
                <ViewCell>
                    <ViewCell.View>
                        <StackLayout Margin="10" Orientation="Vertical"
                                     Padding="10">
                            <Label Text="Full name:"/>
                            <Entry Text="{Binding FullName}"/>
                            <Label Text="Date of birth:"/>
                            <DatePicker Date="{Binding DateOfBirth,Mode=Two
Way}"/>
                            <Label Text="Address:"/>
                            <Entry Text="{Binding Address}"/>
                        </StackLayout>
                    </ViewCell.View>
                </ViewCell>
            </DataTemplate>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ListView x:Name="PeopleList" VerticalOptions="FillAndExpand"
              HasUnevenRows="True" ItemTemplate="{StaticResource MyTemplate
}"/>
</ContentPage>
```

You can also disable item selection with the **`SelectionMode = "None"`** property assignment. This can be useful when displaying read-only data.

## Working with the TableView

When you need to present a list of settings, data in a form, or data that is different from row to row, you can consider the **`TableView`** control. The **`TableView`** is based on sections and can display content through the same cells described previously. With this view, you need to specify a value for its **`Intent`** property, which basically represents the type of information you need to display. Possible values are **`Settings`** (list of settings), **`Data`** (to display data entries), **`Form`** (when the table view acts like a form), and **`Menu`** (to present a menu of selections). Code Listing 21 provides an example.

*Code Listing 21*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <ContentPage.Content>
        <TableView Intent="Settings">
            <TableRoot>
                <TableSection Title="Network section">
                    <SwitchCell Text="Allowed" On="True"/>
                </TableSection>
                <TableSection Title="Push notifications">
                    <SwitchCell Text="Allowed" On="True"/>
                </TableSection>
            </TableRoot>
        </TableView>
    </ContentPage.Content>
</ContentPage>
```

You can divide the **TableView** into multiple **TableSection**s. Each contains a cell to display the required type of information, and of course, you can use a **ViewCell** for a custom, more complex template. Figure 52 shows an example of **TableView** based on the previous listing.

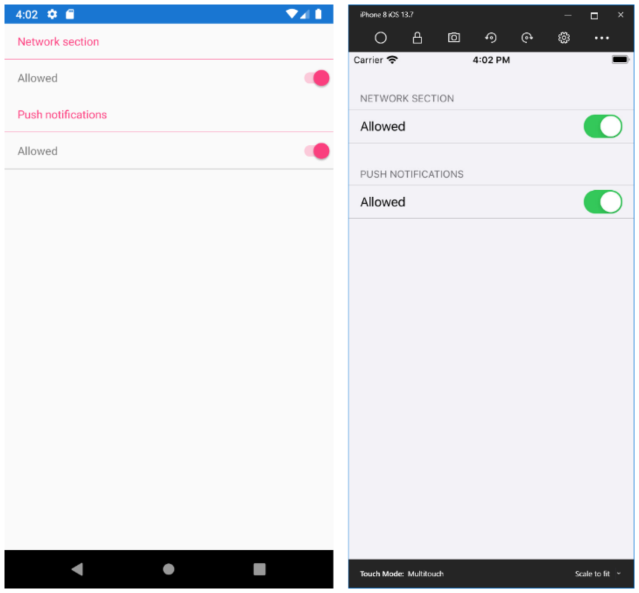


*Figure 52: A TableView in action*

Obviously, you can bind cell properties to objects rather than setting their value explicitly like in the previous example.

## Showing and selecting values with the Picker view

With mobile apps, it is common to provide the user with an option to select an item from a list of values, which can be accomplished with the **Picker** view. Xamarin.Forms 2.3.4 has introduced data-binding support in the **Picker**. You can now easily bind a **List<T>** or **ObservableCollection<T>** to its **ItemsSource** property and retrieve the selected item via its **SelectedItem** property.

For example, suppose you have the following **Fruit** class:

```
public class Fruit
{
    public string Name { get; set; }
    public string Color { get; set; }
}
```

Now, in the user interface, suppose you want to ask the user to select a fruit from a list with the XAML shown in Code Listing 22.

*Code Listing 22*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
```

```
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App2"
             x:Class="App2.MainPage">

    <ContentPage.Content>

        <StackLayout VerticalOptions="FillAndExpand">
            <Label Text="Select your favorite fruit:"/>
            <Picker x:Name="FruitPicker" ItemDisplayBinding="{Binding Name}
"
                    SelectedIndexChanged="FruitPicker_SelectedIndexChanged"
/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

As you can see, the **`Picker`** exposes the **`SelectedIndexChanged`** event, which is raised when the user selects an item. With the **`ItemDisplayBinding`**, you specify which property from the bound object it needs to display: in this case, the fruit name. The **`ItemsSource`** property can, instead, be assigned either in XAML or in the code-behind. In this case, a collection can be assigned in C#, as demonstrated in Code Listing 23.

*Code Listing 23*

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        var apple = new Fruit { Name = "Apple", Color = "Green" };
        var strawberry = new Fruit { Name = "Strawberry", Color = "Red" };
        var orange = new Fruit { Name = "Orange", Color = "Orange" };

        var fruitList = new ObservableCollection<Fruit>()
            { apple, strawberry, orange };
        this.FruitPicker.ItemsSource = fruitList;
    }

    private async void FruitPicker_SelectedIndexChanged(object sender,
                                                        EventArgs e)
    {
        var currentFruit = this.FruitPicker.SelectedItem as Fruit;
        if (currentFruit != null)
            await DisplayAlert("Selection",
            $"You selected {currentFruit.Name}", "OK");
    }
}
```

Like the same-named property in the **ListView**, **ItemsSource** is of type **object** and can bind to any object that implements the **IEnumerable** interface. Notice how you can retrieve the selected item handling the **SelectedIndexChanged** event and casting the **Picker.SelectedItem** property to the type you expect. In such situations, it is convenient to use the **as** operator, which returns null if the conversion fails, instead of an exception. Figure 53 shows how the user can select an item from the picker.

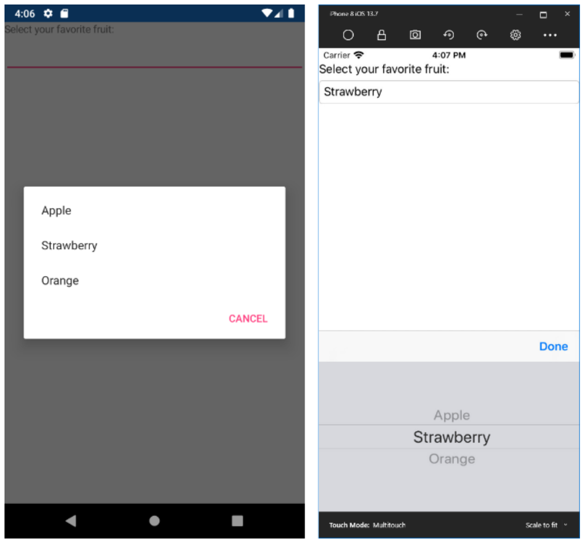


*Figure 53: Selecting items with a Picker*

Data-binding support was added to the **Picker** only with Xamarin.Forms 2.3.4. In previous versions, you could only manually populate its **Items** property via the **Add** method, and then handle indices. This is the real reason why the **SelectedIndexChanged** event exists, but it is still useful with the new approach. Data binding a list to the **Picker** is very common, but you can certainly still populate the list manually and handle the index.

## Binding images

Displaying images in data-binding scenarios is very common, and Xamarin.Forms makes it easy to do. You simply need to bind the **Image.Source** property to an object of type **ImageSource**, or to a URL that can be represented by both a **string** and a **Uri**. For example, suppose you have a class with a property that stores the URL of an image as follows:

```
public class Picture
{
      public Uri PictureUrl { get; set; }
}
```

When you have an instance of this class, you can assign the **PictureUrl** property:

```
var picture1 = new Picture();
picture1.PictureUrl = new Uri("http://mystorage.com/myimage.jpg");
```

Supposing you have an **Image** view in your XAML code and a **BindingContext** assigned with an instance of the class, data binding would work as follows:

```
<Image Source="{Binding PictureUrl}"/>
```

XAML has a type converter for the **`Image.Source`** property, so it automatically resolves strings and **`Uri`** instances into the appropriate type.

## Introducing value converters

The last sentence of the previous section about image binding highlights the existence of type converters that resolve specific types into the appropriate type for the **`Image.Source`** property. This actually happens with many other views and types. For example, if you bind an integer value to the **`Text`** property of an **`Entry`** view, such an integer is converted into a string by a XAML type converter. However, there are situations in which you might want to bind objects that XAML type converters cannot automatically convert into the type a view expects.

For example, you might want to bind a **`Color`** value to a **`Label`**'s **`Text`** property, which is not possible out of the box. In these cases, you can create *value converters*. A value converter is a class that implements the **`IValueConverter`** interface, and that must expose the **`Convert`** and **`ConvertBack`** methods. **`Convert`** translates the original type into a type that the view can receive, while **`ConvertBack`** does the opposite.

Code Listing 24 shows an example of a value converter that converts a string containing HTML markup into an object that can be bound to the **`WebView`** control. **`ConvertBack`** is not implemented because this value converter is supposed to be used in a read-only scenario, so a round-trip conversion is not required.

*Code Listing 24*

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Xamarin.Forms;

namespace App1
{
    public class HtmlConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
               object parameter, CultureInfo culture)
        {
            try
            {
                var source = new HtmlWebViewSource();
                string originalValue = (string)value;

                source.Html = originalValue;
                return source;
            }
            catch (Exception)
            {
                return value;
            }
        }

        public object ConvertBack(object value, Type targetType,
                      object parameter, CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

Both methods always receive the data to convert as object instances, and then you need to cast the object into a specialized type for manipulation. In this case, **`Convert`** creates an **`HtmlWebViewSource`** object, converts the received **`object`** into a **`string`**, and populates the **`Html`** property with the string that contains the HTML markup. The value converter must then be declared in the resources of the XAML file where you wish to use it (or in App.xaml). Code Listing 25 provides an example that also shows how to use the value converter.

*Code Listing 25*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <ContentPage.Resources>
        <local:HtmlConverter x:Key="HtmlConverter"/>
    </ContentPage.Resources>

    <!-- Assumes you have a data-bound .NET object that exposes
    a property called HtmlContent. -->
    <WebView Source="{Binding HtmlContent,
     Converter={StaticResource HtmlConverter}}"/>
</ContentPage>
```

You declare the converter as you would do with any other resource. Then your binding will also contain the **`Converter`** expression, which points to the value converter with the typical syntax you used with other resources.

## Displaying lists with the CollectionView

Xamarin.Forms 4.0 has introduced a new view to display lists of data, the **`CollectionView`**. Its biggest benefits are: simpler API surface, no need for custom renderers for common requirements as for the **`ListView`**, and no need for view cells, as you will see shortly. However, the documentation recommends considering the **`CollectionView`** as an alternative to the **`ListView`**, rather than a replacement. Continuing the example shown previously, Code Listing 26 shows how you could bind a collection of **`Person`** objects to a **`CollectionView`**.

*Code Listing 26*

```
<CollectionView x:Name="PeopleList"
 ItemsSource="{Binding People}"
 SelectionMode="Single"
 SelectionChanged="PeopleList_SelectionChanged"
 VerticalScrollBarVisibility="Never"
 HorizontalScrollBarVisibility="Never">

    <CollectionView.ItemTemplate>
        <DataTemplate>
            <StackLayout Margin="10">
                <Label Text="Full name:"/>
                <Entry Text="{Binding FullName}"/>
                <Label Text="Date of birth:"/>
                <DatePicker Date="{Binding DateOfBirth,
                            Mode=TwoWay}"/>
                <Label Text="Address:"/>
                <Entry Text="{Binding Address}"/>
            </StackLayout>
        </DataTemplate>
    </CollectionView.ItemTemplate>
</CollectionView>
```

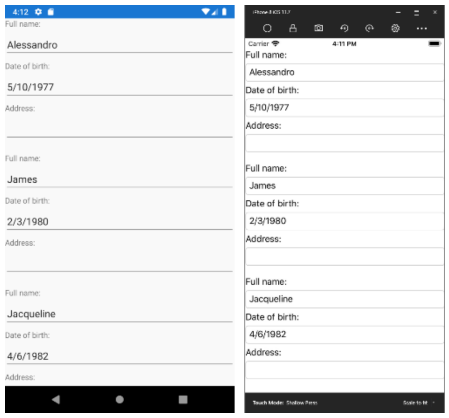
You can control the visibility of the scroll bars directly, without custom renderers, using the **`VerticalScrollBarVisibility`** and **`HorizontalScrollBarVisibility`** properties. The **`SelectionMode`** property allows for improved item selection with values like **`None`**, **`Single`**, and **`Multiple`**. The **`SelectionChanged`** event is fired when an item is selected. Code Listing 27 shows how to work on the item selection, depending on the value of **`SelectionMode`**.

*Code Listing 27*

```
private void PeopleList_SelectionChanged(object sender,
        SelectionChangedEventArgs e)
{
    // In case of single selection
    var selectedPerson = this.PeopleList.SelectedItem as Person;

    // In case of multi-selection:
    var singlePerson = e.CurrentSelection.FirstOrDefault()
```

```
                as Person;

            var selectedObjects = e.CurrentSelection.Cast<Person>();
            foreach (var person in selectedObjects)
            {
                // Handle your object properties here...
            }
        }
```

Figure 54 shows the result of these code listings.


Full name:
Alessandro
Date of birth:
5/10/1977
Address:

Full name:
James
Date of birth:
2/3/1980
Address:

Full name:
Jacqueline
Date of birth:
4/6/1982
Address:

*Figure 54: Displaying lists with the CollectionView*

The **`CollectionView`** also makes it very simple to customize its layout. By default, the orientation is vertical, but you can use the **`LinearItemsLayout`** property to change the orientation to **`Horizontal`** as follows:

```
<CollectionView.ItemsLayout>
    <LinearItemsLayout Orientation="Horizontal"/>
</CollectionView.ItemsLayout>
```

The **`CollectionView`** also supports a grid view layout, which can be accomplished with the **`GridItemsLayout`** property as follows:

```
<CollectionView.ItemsLayout>
    <GridItemsLayout Orientation="Horizontal" Span="3"/>
</CollectionView.ItemsLayout>
```

The **`Span`** property indicates how many items are visible per line of the grid view. The **`CollectionView`** has another amazing property called **`EmptyView`**, which you use to display a specific view if the bound collection is empty (null or zero elements). For example, the following code demonstrates how to show a red label if the bound collection has no data:

```
<CollectionView.EmptyView>
    <Label Text="No data is available" TextColor="Red" FontSize="Large"/>
</CollectionView.EmptyView>
```

There is no similar option in the **ListView**, and it's your responsibility to implement and manage the empty data scenario. You can also display complex views with empty data by using the **EmptyViewTemplate** property instead, which works like in the following example:

```
<CollectionView.EmptyViewTemplate>
    <DataTemplate>
        <StackLayout Orientation="Vertical" Spacing="20">
            <Image Source="EmptyList.png" Aspect="Fill"/>
            <Label Text="No data is available" TextColor="Red"/>
        </StackLayout>
    </DataTemplate>
</CollectionView.EmptyViewTemplate>
```

In this code snippet, an image and a text message are displayed vertically inside a **StackLayout** view. Figure 55 shows an example of an empty view.

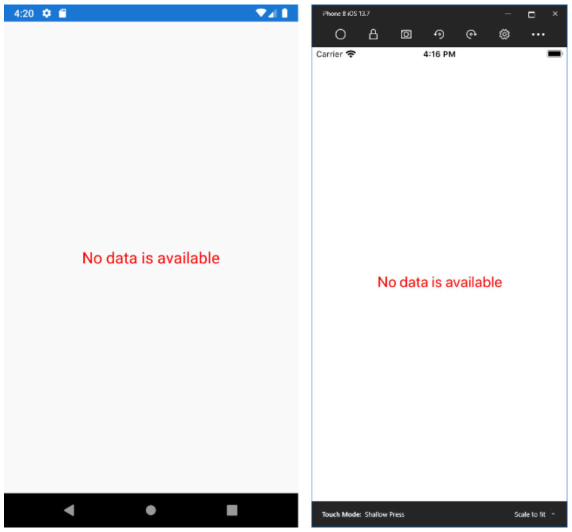


*Figure 55: Default empty view*

The **CollectionView** actually has no built-in support for the pull-to-refresh gesture; however, a new view called **RefreshView** allows you to implement this common functionality. This view requires a bit of knowledge of the Model-View-ViewModel pattern, so it's discussed in the next section.

## Scrolling lists with the CarouselView

The **CarouselView** is a control that allows for scrolling lists. Typically, a **CarouselView** is used to scroll lists horizontally, but it also supports vertical orientation. Actually, Xamarin.Forms already had a **CarouselView** in its codebase, but in the lastest versions it has been completely redesigned, and it's now built on top of the **CollectionView**.

When you declare a **CarouselView**, you will still define a **DataTemplate** like for the **CollectionView**, and you can also define the **EmptyView** and **EmptyViewTemplate** properties. Code Listing 28 shows an example based on a collection of **Person** objects, like for the **CollectionView**.

*Code Listing 28*

```
        <CarouselView x:Name="PeopleList" ItemsSource="{Binding People}"
                      CurrentItemChanged=
                      "PeopleList_CurrentItemChanged"
                      PositionChanged="PeopleList_PositionChanged"
                      CurrentItem="{Binding SelectedPerson}">
            <CarouselView.ItemsLayout>
                <LinearItemsLayout Orientation="Horizontal"
                     SnapPointsAlignment="Center"
                     SnapPointsType="Mandatory"/>
            </CarouselView.ItemsLayout>
            <CarouselView.ItemTemplate>
                <DataTemplate>
                    <StackLayout Margin="10">
                        <Label Text="Full name:"/>
                        <Entry Text="{Binding FullName}"/>
                        <Label Text="Date of birth:"/>
                        <DatePicker Date="{Binding DateOfBirth,
                                    Mode=TwoWay}"/>
                        <Label Text="Address:"/>
                        <Entry Text="{Binding Address}"/>
                    </StackLayout>
                </DataTemplate>
            </CarouselView.ItemTemplate>
        </CarouselView>
```

The **`PositionChanged`** event is fired when the control is scrolled. The **`PositionChangedEventArgs`** object provides the **`CurrentPosition`** and **`PreviousPosition`** properties, both of type **`int`**. The **`CurrentItemChanged`** event is fired when the user taps on a displayed item. The **`CurrentItemChangedEventArgs`** object provides the **`CurrentItem`** and **`PreviousItem`** properties, both of type object. Notice how the **`CurrentItem`** property is data-bound to an individual instance of the **`Person`** class, and this simplifies the way you get the instance of the currently selected object in the **`CarouselView`**.

In the layout definition, the **`SnapPointsAlignment`** property specifies how snap points are aligned with items, whereas the **`SnapPointsType`** property specifies the behavior of snap points when scrolling. Figure 56 shows an example of the **`CarouselView`** in action, with items scrolling horizontally.

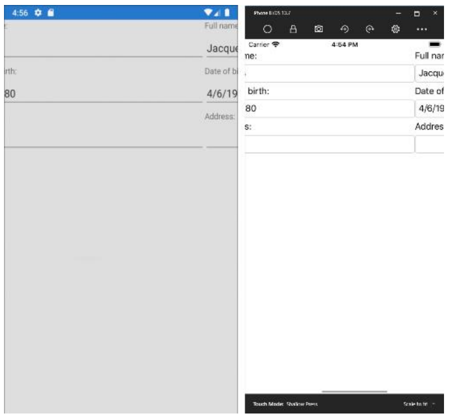


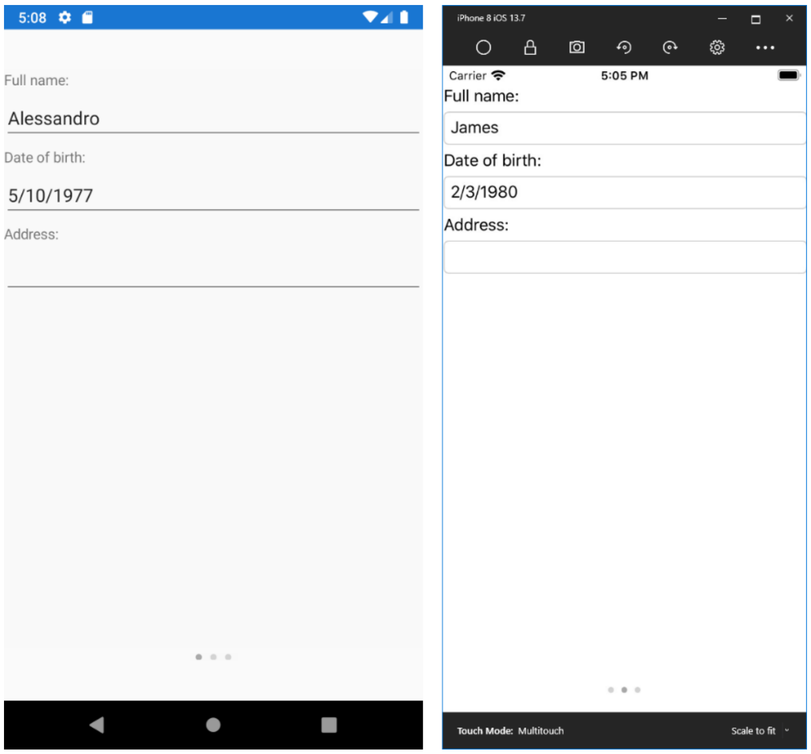
*Figure 56: Scrolling lists with the CarouselView*

## Introducing the IndicatorView

In recent times, Xamarin.Forms introduced the **IndicatorView**, a control that displays indicators that represent the number of items, and current position, in a **CarouselView**. The latter has an **IndicatorView** property, which you assign with the name of the related **IndicatorView**, usually declared below the **CarouselView**. Continuing the example of the previous paragraph, you would extend the XAML as follows:

```
 <StackLayout Orientation="Vertical">

    <CarouselView IndicatorView="PersonIndicatorView">

    ...

    </CarouselView>

    <IndicatorView x:Name="PersonIndicatorView"

           IndicatorColor="LightGray"

           SelectedIndicatorColor="DarkGray"

           HorizontalOptions="Center" />

</StackLayout>
```

You will need to wrap both views into a layout to have them close in the user interface. The **IndicatorColor** and **SelectedIndicatorColor** properties represent the color of the indicator in its normal and selected states, respectively. You can also use the **IndicatorsShape** property and set it with **Square** (default) or **Circle** to define the appearance of the indicators. The **IndicatorSize** property allows you to customize the indicator size, but you can also provide your completely custom appearance using the **IndicatorTemplate** property, in which you can supply views to provide your own representation inside a **DataTemplate**. Figure 57 shows an example of how the **IndicatorView** looks, at the bottom of the page.



Full name:
Alessandro
Date of birth:
5/10/1977
Address:

Full name:
James
Date of birth:
2/3/1980
Address:

*Figure 57: Assigning an IndicatorView to a CarouselView*

## Introducing Model-View-ViewModel

Model-View-ViewModel (MVVM) is an architectural pattern used in XAML-based platforms that allows for clean separation between the data (model), the logic (view model), and the user interface (view). With MVVM, pages only contain code related to the user interface, they strongly rely on data binding, and most of the work is done in the view model. MVVM can be quite complex if you have never seen it before, so I will try to simplify the explanations as much as possible, but you should use Xamarin's MVVM documentation as a reference.

Let's start with a simple example and a fresh Xamarin.Forms solution based on the .NET Standard code-sharing strategy. Imagine you want to work with a list of `Person` objects. This is your model, and you can reuse the `Person` class from before. Add a new folder called **Model** to your project and add a new **Person.cs** class file to this folder, pasting in the code of the `Person` class. Next, add a new folder called **ViewModel** to the project and add a new class file called **PersonViewModel.cs**.

Before writing the code for it, let's summarize some important considerations:

- The view model contains the business logic, acts like a bridge between the model and the view, and exposes properties to which the view can bind.
- Among such properties, one will certainly be a collection of `Person` objects.
- In the view model, you can load data, filter data, execute save operations, and query data.

Loading, filtering, saving, and querying data are examples of actions a view model can execute against data. In a classic development approach, you would handle **`Clicked`** events on **`Button`** views and write the code that executes an action. However, in MVVM, views should only contain code related to the user interface, not code that executes actions against data. In MVVM, view models expose the so-called *commands*. A command is a property of type **`ICommand`** that can be data-bound to views such as **`Button`**, **`SearchBar`**, **`CollectionView`**, **`ListView`**, and **`TapGestureRecognizer`** objects. In the UI, you bind a view to a command in the view model. In this way, the action is executed in the view model instead of in the view's code behind. Code Listing 29 shows the **`PersonViewModel`** class definition.

*Code Listing 29*

```
using MvvmSample.Model;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Runtime.CompilerServices;
using System.Windows.Input;
using Xamarin.Forms;

namespace MvvmSample.ViewModel
{
    public class PersonViewModel: INotifyPropertyChanged
    {
        public ObservableCollection<Person> People { get; set; }

        private Person _selectedPerson;
        public Person SelectedPerson
        {
            get
            {
                return _selectedPerson;
            }
            set
            {
                _selectedPerson = value;
                OnPropertyChanged();
            }
        }

        public ICommand AddPersonCommand { get; set; }
        public ICommand DeletePersonCommand { get; set; }

        public event PropertyChangedEventHandler PropertyChanged;
        private void OnPropertyChanged([CallerMemberName] string
propertyName
                                        = null)
        {
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(propertyName));
        }

        private void LoadSampleData()
        {
            People = new ObservableCollection<Person>();

            // sample data
            Person person1 =
                new Person
                {
                    FullName = "Alessandro",
                    Address = "Italy",
                    DateOfBirth = new DateTime(1977, 5, 10)
                };
            Person person2 =
                new Person
                {
                    FullName = "Robert",
                    Address = "United States",
                    DateOfBirth = new DateTime(1960, 2, 1)
                };
            Person person3 =
                new Person
                {
                    FullName = "Niklas",
                    Address = "Germany",
                    DateOfBirth = new DateTime(1980, 4, 2)
                };

            People.Add(person1);
            People.Add(person2);
            People.Add(person3);
        }

        public PersonViewModel()
        {
            LoadSampleData();

            AddPersonCommand =
                new Command(() => People.Add(new Person()));

            DeletePersonCommand =
                new Command<Person>((person) => People.Remove(person));
        }
    }
}
```

The **People** and **SelectedPerson** properties expose a collection of **Person** objects and a single **Person**, respectively, and the latter will be bound to the **SelectedItem** property of a **ListView**, as you will see shortly. Notice how properties of type **ICommand** are assigned with instances of the **Command** class, to which you can pass an **Action** delegate via a lambda expression that executes the desired operation. The **Command** provides an out-of-the-box implementation of the **ICommand** interface, and its constructor can also receive a parameter, in which case you must use its generic overload (see **DeletePerson** assignment). In that case, the **Command** works with objects of type **Person**, and the action is executed against the received object. Commands and other properties are data-bound to views in the user interface.

***Note: Here I demonstrated the most basic use of commands. However, commands also expose a*** *CanExecute* ***Boolean method that determines whether an action can be executed or not. Additionally, you can create custom commands that implement*** *ICommand* ***and must explicitly implement the*** *Execute* ***and*** *CanExecute* ***methods, where*** *Execute* ***is invoked to run an action. For further details, look at the official documentation.***

Now it is time to write the XAML code for the user interface. Code Listing 30 shows how to use the new **CollectionView** for this, and how to bind two **Button** views to commands.

*Code Listing 30*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MvvmSample.MainPage">

    <StackLayout>
        <CollectionView x:Name="PeopleList"
                        ItemsSource="{Binding People}"
                        SelectionMode="Single"
                        SelectedItem="{Binding SelectedPerson}"
                        VerticalScrollBarVisibility="Never"
                        HorizontalScrollBarVisibility="Never">

            <CollectionView.ItemTemplate>
               <DataTemplate>
                    <StackLayout Margin="10">
                       <Label Text="Full name:"/>
                       <Entry Text="{Binding FullName}"/>
                       <Label Text="Date of birth:"/>
                       <DatePicker Date="{Binding DateOfBirth,
                                   Mode=TwoWay}"/>
                       <Label Text="Address:"/>
                       <Entry Text="{Binding Address}"/>
                    </StackLayout>
                </DataTemplate>
            </CollectionView.ItemTemplate>
         </CollectionView>
         <StackLayout Orientation="Horizontal">
             <Button Text="Add" Command="{Binding AddPerson}"/>
             <Button Text="Delete" Command="{Binding DeletePerson}"
                     CommandParameter=
                     "{Binding Source={x:Reference PeopleList},
                     Path=SelectedItem}"/>
             <Button Text="Detail" Command="{Binding ViewPersonDetail}"/>
         </StackLayout>
     </StackLayout>
 </ContentPage>
```

Notice how:

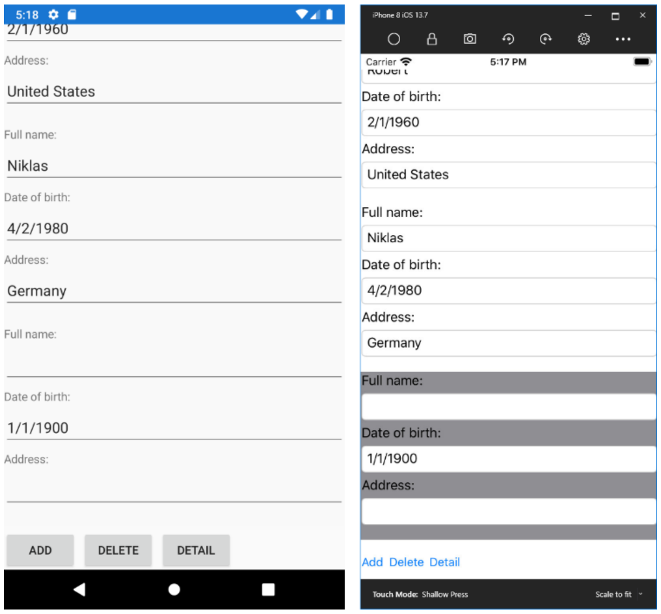
- The **CollectionView.ItemsSource** property is bound to the **People** collection in the view model.
- The **CollectionView.SelectedItem** property is bound to the **SelectedPerson** property in the view model.
- The first **Button** is bound to the **AddPerson** command in the view model.
- The second **Button** is bound to the **DeletePerson** command, and it passes the selected **Person** object in the **CollectionView** with a special binding expression; **Source** represents the data source, in this case the **CollectionView**, referred to with **x:Reference**; and **Path** points to the property in the source that exposes the object you want to pass to the command as a parameter (simply referred to as command parameter).

The final step is to create an instance of the view model and assign it to the **BindingContext** of the page, which you can do in the page code-behind, as demonstrated in Code Listing 31.

*Code Listing 31*

```
using MvvmSample.ViewModel;
using Xamarin.Forms;

namespace MvvmSample
{
    public partial class MainPage : ContentPage
    {
        // Not using a field here because properties
        // are optimized for data binding.
        private PersonViewModel ViewModel { get; set; }

        public MainPage()
        {
            InitializeComponent();

            this.ViewModel = new PersonViewModel();
            this.BindingContext = this.ViewModel;
        }
    }
}
```

If you now run the application (see Figure 54), you will see the list of **`Person`** objects, and you will be able to use the two buttons. The real benefit is that the whole logic is in the view model. With this approach, if you change the logic in the properties or in the commands, you will not need to change the page code. In Figure 58, you can see a new **`Person`** object added via command binding.



2/1/1960
Address:
United States
Full name:
Niklas
Date of birth:
4/2/1980
Address:
Germany
Full name:
Date of birth:
1/1/1900
Address:

Robert
Date of birth:
2/1/1960
Address:
United States
Full name:
Niklas
Date of birth:
4/2/1980
Address:
Germany
Full name:
Date of birth:
1/1/1900
Address:

*Figure 58: Showing a list of people and adding a new person with MVVM*

MVVM is very powerful, but real-world implementations can be very complex. For example, if you want to navigate to another page and you have commands, the view model should contain code related to the user interface (launching a page) that does not adhere to the principles of MVVM. Obviously, there are solutions to this problem that require further knowledge of the pattern, so I recommend you look at books and articles on the internet for further study. There's no need to reinvent the wheel: many robust and popular MVVM libraries already exist, and you might want to choose one from among the following:

- Prism
- MVVM Light Toolkit
- FreshMvvm
- MvvmCross

I have personally worked with FreshMvvm, but all the aforementioned alternatives are powerful enough to save you a lot of time.

## Implementing pull-to-refresh: The RefreshView

The **`CollectionView`** provides many benefits due to a simplified API surface, but it lacks built-in support for the pull-to-refresh gesture. Fortunately, Xamarin.Forms has introduced the **`RefreshView`**, a container that allows for implementing the pull-to-refresh gesture where not available. This view is discussed in this paragraph because, as you will see shortly, it requires knowledge of how commands work.

***Tip: The `RefreshView` can be used with any view that implements scrolling, not only the `CollectionView`. For instance, a `ScrollView` with child elements can now implement pull-to-refresh if put inside a `RefreshView`.***

Let's start by adding the **`RefreshView`** in the XAML. Continuing the current code example, you will wrap the **`CollectionView`** inside a **`RefreshView`** as follows:

```
    <StackLayout>
        <RefreshView RefreshColor="Teal"
                     IsRefreshing="{Binding IsRefreshing}"
                     Command="{Binding RefreshCommand}">
            <CollectionView x:Name="PeopleList" … >
            …
            </CollectionView>
        </RefreshView>
    </StackLayout>
```

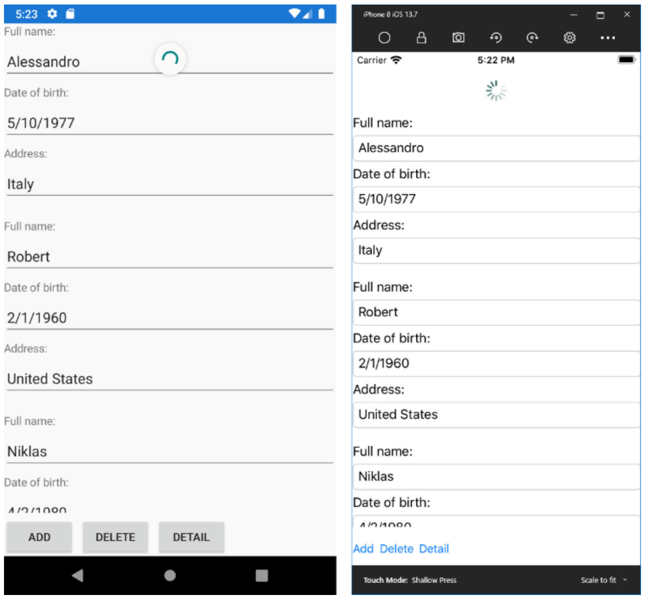
The **`RefreshView`** exposes a nice property called **`RefreshColor`**, which allows you to change the color of the spinner that is displayed while the pull-to-refresh is active quickly. The **`IsRefreshing`** property is bound to a bool property in the view model, which enables or disables the pull-to-refresh spinner when **`true`** or **`false`**, respectively. The **`Command`** property is bound to a command object that performs the actual refresh operation. The **`IsRefreshing`** property and the **`RefreshCommand`** property will be defined in the view model as follows:

```
private bool _isRefreshing;
public bool IsRefreshing
{
    get
    {
        return _isRefreshing;
    }
    set
    {
        _isRefreshing = value;
        OnPropertyChanged();
    }
}
public ICommand RefreshCommand { get; set; }
```

In the constructor of the view model, you can then assign the **`RefreshCommand`** with the action that will be performed to refresh the data, so add the following code:

```
RefreshCommand =
    new Command(async () =>
    {
        IsRefreshing = true;
        LoadSampleData();
        // Simulates a longer operation
        await Task.Delay(2000);
        IsRefreshing = false;
    }
);
```

The action does simple work: it assigns **`IsRefreshing`** with **`true`** in order to activate the pull-to-refresh user interface, it reloads the sample data, and then it assigns **`IsRefreshing`** with **`false`** when finished, to disable the pull-to-refresh user interface. Now you can run the sample, pull and release the list of data, and see how the gesture is implemented. Figure 59 shows an example.



Full name:
Alessandro
Date of birth:
5/10/1977
Address:
Italy
Full name:
Robert
Date of birth:
2/1/1960
Address:
United States
Full name:
Niklas
Date of birth:

*Figure 59: Pull-to-refresh with the RefreshView*

Obviously, in real-world scenarios your data will take a couple of seconds to reload, so you will not need to use the **`Task.Delay`** method to simulate a long-running operation.

## Chapter summary

XAML plays a fundamental role in Xamarin.Forms and allows for defining reusable resources and for data-binding scenarios. Resources are reusable styles, data templates, and references to objects you declare in XAML. In particular, styles allow you to set the same properties to all views of the same type, and they can extend other styles with inheritance. XAML also includes a powerful data-binding engine that allows you to bind objects quickly to visual elements in a two-way communication flow.

In this chapter, you have seen how to bind a single object to individual visual elements and collections of objects to the **`ListView`** and **`CollectionView`**. You have seen how to define data templates so that the **`ListView`** and the **`CollectionView`** can have knowledge of how items must be presented, and you have learned about value converters, special objects that come in to help when you want to bind objects of a type that is different from the type a view supports.

In the second part of the chapter, you walked through an introduction to the Model-View-ViewModel pattern, focusing on separating the logic from the UI and understanding new objects and concepts such as commands. Finally, you have seen how to implement pull-to-refresh quickly with the new **`RefreshView`**. In the next chapter, you will walk through new, exciting features: brushes and shapes.

# Chapter 8 Brushes and Shapes

If you have ever worked with professional designers on real-world mobile apps, you know how frustrating it was to say, “We cannot do this” when they asked to include shapes and gradient colors in the app design. There were options, like third-party libraries, but this creates a dependency, and it’s not always allowed by some companies. Luckily enough, Xamarin.Forms 5.0 offers brushes and shapes, providing the ultimate solution to this kind of problem.

## Understanding brushes

Brushes are new types in Xamarin.Forms that expand the way you can add colors to your visual elements. Before Xamarin.Forms 4.8, you could only use solid colors; now you can use linear and radial gradients as well. If you have worked with Windows Presentation Foundation (WPF) and native UWP in the past, you know what I’m talking about. Consider the following line, which declares a `Button` with a background color:

```
<Button Text="Tap here" BackgroundColor="LightBlue"/>
```

The `BackgroundColor` is of type `Color` and allows for assigning an individual color. Now a new property called `Background` is available, and it is of type `Brush`. This is an abstract type definition for specialized types like `SolidColorBrush`, `LinearGradientBrush`, and `RadialGradientBrush`. Let’s discuss these in more detail.

### Solid colors with SolidColorBrush

As the name implies, `SolidColorBrush` assigns a solid color to a property of type `Brush`. This can be done inline, like in the previous button declaration, which takes advantage of type converters. The value can be one of the literals displayed by the IntelliSense, or even a color in HEX format (for example, #FFFFFF). A `SolidColorBrush` can be assigned with two different syntaxes. The first one is an inline assignment, as follows:

```
<Frame CornerRadius="5" Background="Green"
       Margin="20" WidthRequest="150"
       HeightRequest="150">
</Frame>
```

The second syntax is the so-called property syntax, with an extended form that will be very useful with gradients:

```
<Frame CornerRadius="5"

       Margin="20" WidthRequest="150"

       HeightRequest="150">
```

```
                <Frame.Background>

                    <SolidColorBrush Color="Green"/>

                </Frame.Background>

            </Frame>
```

As I mentioned previously, you can use not only color literals, but also hexadecimal codes.

***Tip: In practice, using a solid color with a property of type*** `Color`***, or with a property of type*** `Brush`***, makes no difference. However, you might want to use properties of type*** `Brush` ***if you plan to replace a solid color brush value with a linear or radial gradient at runtime, since both these derive from*** `Brush`***, like*** `SolidColorBrush`***.***

## Linear gradients with LinearGradientBrush

You can create linear gradients with the **`LinearGradientBrush`** object. Gradients can be horizontal, vertical, and diagonal. Linear gradients have two-dimensional coordinates, represented by the **`StartPoint`** and **`EndPoint`** properties on an axis where a value of 0,0 is the top-left corner of the view, and 1,1 is the bottom-right corner of the view. The default value for StartPoint is **`0,0`**, while the default value for **`EndPoint`** is **`1,1`**. The following example draws a horizontal linear gradient as the background for a **`Frame`** view:

```
<Frame CornerRadius="5"
       Margin="20" WidthRequest="150"
       HeightRequest="150">
    <Frame.Background>
        <!-- StartPoint defaults to (0,0) -->
        <LinearGradientBrush EndPoint="1,0">
            <GradientStop Color="Blue"
              Offset="0.1" />
            <GradientStop Color="Violet"
              Offset="1.0" />
        </LinearGradientBrush>
    </Frame.Background>
</Frame>
```

Each **`GradientStop`** represents a color and its position in the gradient via the **`Offset`** property. Figure 60 shows how the gradient looks.

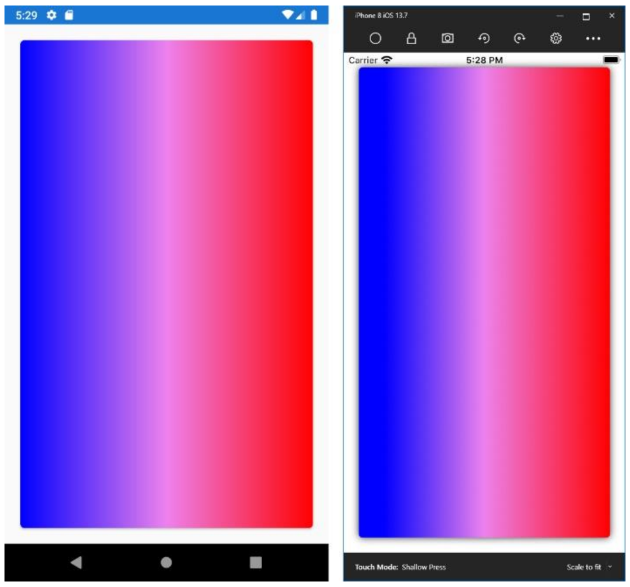


*Figure 60: A horizontal linear gradient*

You can change the orientation of the gradient to vertical by replacing the value of the **`EndPoint`** property as follows:

```
<LinearGradientBrush EndPoint="0,1">
```

You can create a diagonal gradient by simply removing **`StartPoint`** and **`EndPoint`** and using their default values of **`0,0`** and **`1,1`**, respectively. You can also add more than two colors. The following code demonstrates this:

```
<LinearGradientBrush EndPoint="1,0">
    <GradientStop Color="Blue"
      Offset="0.1" />
    <GradientStop Color="Violet"
      Offset="0.5" />
    <GradientStop Color="Red"
      Offset="1.0" />
</LinearGradientBrush>
```

Notice how the **`Offset`** property is used to determine the position of the colors; in this case, **`Violet`** is placed in the middle of the gradient in a scale between 0.1 and 1.

## Circular gradients with RadialGradientBrush

You can create circular gradients using the **`RadialGradientBrush`** object. This object exposes the **`Radius`** property, of type **`double`**, which represents the radius of the circle for the gradient, and whose default value is **`0.5`**. It also exposes the **`Center`** property, of type **`Point`**, which represents the center of the circle in the gradient, and whose default value is **`0.5,0.5`**. In its simplest form, a **`RadialGradientBrush`** can be declared as follows:

```
<Frame CornerRadius="5"
       Margin="20" WidthRequest="150"
       HeightRequest="150">
    <Frame.Background>
        <!-- Radius defaults to (0.5,0.5) -->
        <RadialGradientBrush>
            <GradientStop Color="Blue"
              Offset="0.1" />
            <GradientStop Color="Violet"
              Offset="1.0" />
        </RadialGradientBrush>
    </Frame.Background>
</Frame>
```

The resulting gradient looks like Figure 61.

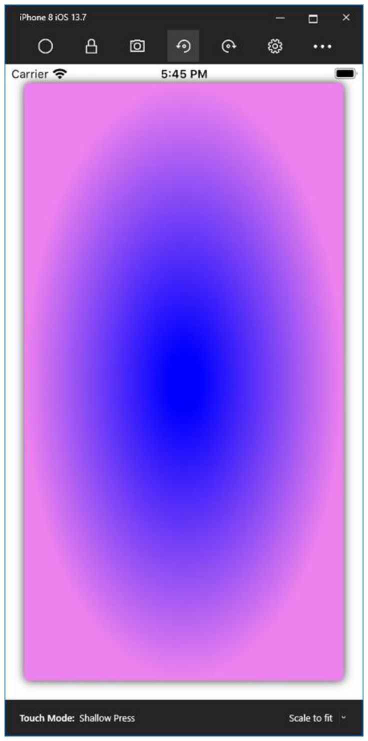


*Figure 61: Drawing a circular gradient*

You can move the center of the gradient with the **`Center`** property. The following value moves the center of the gradient to the top-left corner of the view:

```
<RadialGradientBrush Center="0.0,0.0">
```

The following assignment moves the center of the gradient to the bottom-right corner of the view:

```
<RadialGradientBrush Center="1.0,1.0">
```

In real world development, you will probably use linear gradients more often, but radial gradients are certainly a very nice addition to the new drawing possibilities.

## Drawing shapes

Xamarin.Forms 4.7 introduced *shapes*, which are views that derive from the **`Shape`** class. Xamarin.Forms offers the following shapes, each with a self-explanatory name: **`Ellipse`**, **`Rectangle`**, **`Line`**, and **`Polygon`**. In addition, a shape called **`Polyline`** allows for drawing custom polygons, and another shape called **`Path`** allows for drawing completely custom shapes. All these shapes share some properties; the most important ones are described in Table 8.

*Table 8: Shapes' properties*

| Property | Type | Description |
|---|---|---|
| **`Aspect`** | **`Stretch`** | Defines how the shape fills its allocated space. Supported values are **`None`**, **`Fill`**, **`Uniform`**, and **`UniformToFill`**. |
| **`Fill`** | **`Brush`** | The brush used to fill the shape's interior. |
| **`Stroke`** | **`Brush`** | The brush used for the shape's outline. |
| **`StrokeThickness`** | **`double`** | The width of the shape's outline (default is 0). |
| **`StrokeDashArray`** | **`DoubleCollection`** | A collection of values that indicate the pattern of dashes and gaps used to outline a shape. |
| **`StrokeDashOffset`** | **`double`** | The distance between dashes. |

***Note: The following examples draw shapes' outlines for a better understanding, but this is totally optional.***

Once you learn the properties listed in Table 8, you will see how shapes are very easy to handle.

## Drawing ellipses and circles

The first shape we discuss is the **`Ellipse`**, which you use to draw ellipses and circles. The following code provides an example where the ellipse is filled in violet and the outline is green, with a thickness of 3:

```
<Ellipse Fill="Violet" Stroke="Green" StrokeThickness="3"
 WidthRequest="250" HeightRequest="100" HorizontalOptions="Center"
 Margin="0,50,0,0"/>
```

The resulting shape is the first one on top shown in Figure 62.

***Tip: Properties like `Fill` are of type `Brush`, but here they are assigned with values of type `Color`. This is because Xamarin is using a built-in type converter.***

## Drawing rectangles

The second shape we'll discuss is the **`Rectangle`**. While keeping an eye on Table 7 for the properties used to draw dashes on the outline, the following example shows how you can use brushes for filling a shape:

```
<Rectangle Stroke="Green" StrokeThickness="4" StrokeDashArray="1,1"
   StrokeDashOffset="6" WidthRequest="250" HeightRequest="100"
   Margin="0,50,0,0" HorizontalOptions="Center">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="LightBlue" Offset="0"/>
            <GradientStop Color="Blue" Offset="0.5"/>
            <GradientStop Color="Violet" Offset="1"/>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

The resulting shape is the second one from top you can see in Figure 62.

## Drawing lines

The next shape we discuss is the **`Line`**. Let's start with some code:

```
<Line X1="0" Y1="30" X2="250" Y2="20" StrokeLineCap="Round" Stroke="Violet"

 StrokeThickness="12" Margin="0,50,0,0" HorizontalOptions="Center"/>
```

**X1** and **X2** represent the starting and ending points of the line on the x-axis. Y1 and Y2 represent the lowest and the highest points on the y-axis, respectively. The **StrokeLineCap** property, of type **PenCap**, describes a shape at the end of a line, and can be assigned with **Flat** (default), **Square**, or **Round**. **Flat** basically draws no shape, **Square** draws a rectangle with the same height and thickness of the line, and **Round** draws a semicircle with the diameter equal to the line's thickness. The resulting shape is the third from the top shown in Figure 62.

## Drawing polygons

Polygons are complex shapes, and Xamarin.Forms provides the **Polygon** class to draw these geometries. The following example draws a triangle filled in violet and with a green, dashed outline:

```
<Polygon Points="50,20 80,60 20,60" Fill="Violet" Stroke="Green"
 StrokeThickness="4" StrokeDashArray="1,1" StrokeDashOffset="6"
 HorizontalOptions="Center"/>
```

The resulting shape is the bottom shape in Figure 62.

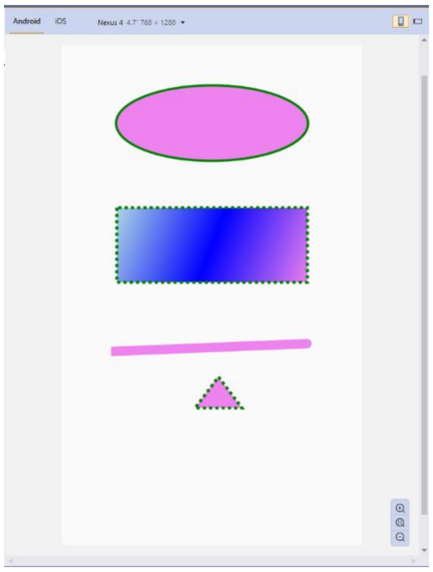


*Figure 62: Drawing basic shapes*

The key property in the **Polygon** shape is **Points**, a collection of **Point** objects, each representing the coordinates of a specific delimiter in the polygon. Because there's basically no limit to the collection of points, you can create very complex polygons.

## Drawing complex shapes with the Polyline

The **Polyline** is a particular shape that allows for drawing a series of straight lines connected to one another, but where the last line does not connect with the first point of the shape. Consider the following example:

```
<Polyline Margin="0,50,0,0" HorizontalOptions="Center"
    Points="0 48, 0 144, 96 150, 100 0, 192 0, 192 96, 50 96, 48 192, 150 200
144 48"
    Fill="Violet" Stroke="DarkGreen" StrokeThickness="3"/>
```

Like the **`Polygon`**, the **`Polyline`** exposes a property called **`Points`**, a collection of **`Point`** objects, each representing the coordinates of a point. The resulting shape is visible in Figure 63.

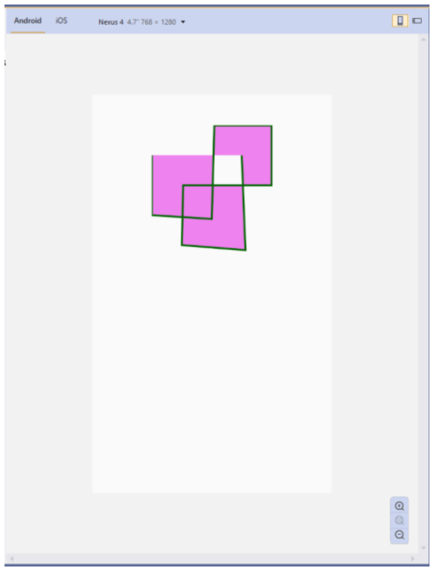


*Figure 63: Drawing complex shapes*

***Note:** `Polyline` exposes a property called `FillRule`, which determines how pieces of the shape should be filled based on the intersection of points. This is not a quick and easy topic and involves some math. For this reason, this property is not discussed here, but you can have a look at the documentation for further details.*

## Hints about Geometries and Paths

The new drawing possibilities are not limited to the basic shapes described in the previous sections. Xamarin.Forms also offers powerful 2D drawings with the so-called *geometries*, and it provides the **`Path`** class that allows drawing curves and complex shapes using geometries. Both are very complex and long topics that cannot fit inside a book of the *Succinctly* series. For this reason, I recommend having a look at the official Geometries and Path documentation pages.

## Chapter summary

This chapter introduced two new features in Xamarin.Forms: brushes and shapes. With brushes, you can assign solid colors (**`SolidColorBrush`**), linear gradients (**`LinearGradientBrush`**) and circular gradients (**`RadialGradientBrush`**) to properties of type **`Brush`**, like views' **`Background`** property. You can then draw shapes with the **`Ellipse`**, **`Rectangle`**, **`Line`**, **`Polygon`**, and **`Polyline`** objects, to which you can assign a brush to the **`Fill`** property as the shape color, and the **`Stroke`** property as the outline color.

So far you have seen what Xamarin.Forms has to offer cross-platform, but there is more power that allows you to use platform-specific APIs. This is the topic of the next chapter.

# Chapter 9  Accessing Platform-Specific APIs

Until now, you have seen what Xamarin.Forms offers in terms of features that are available on each supported platform, walking through pages, layouts, and controls that expose properties and capabilities that will certainly run on Android, iOS, and Windows. Though this simplifies cross-platform development, it is not enough to build real-world mobile applications. In fact, more often than not, mobile apps need to access sensors, the file system, the camera, and the network; send push notifications; and more. Each operating system manages these features with native APIs that cannot be shared across platforms and, therefore, that Xamarin.Forms cannot map into cross-platform objects.

If Xamarin.Forms did not provide a way to access native APIs, it would not be very useful. Luckily, Xamarin.Forms provides multiple ways to access platform-specific APIs that you can use to access practically everything from each platform. Thus, there is no limit to what you can do with Xamarin.Forms. In order to access platform features, you will need to write C# code in each platform project. This is what this chapter explains, together with all the options you have to access iOS, Android, and Windows APIs from your shared codebase.

## The Device class and the OnPlatform method

The **`Xamarin.Forms`** namespace exposes an important class called **`Device`**. This class allows you to detect the platform your app is running on and the device idiom (tablet, phone, desktop). This class is particularly useful when you need to adjust the user interface based on the platform.

The following code demonstrates how to take advantage of the **`Device.RuntimePlatform`** property to detect the running platform and make UI-related decisions based on its value:

```
// Label1 is a Label view in the UI
switch(Device.RuntimePlatform)
{
    case Device.iOS:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Large, Label1);
        break;
    case Device.Android:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Medium, Label1);
        break;
    case Device.WinPhone:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Medium, Label1);
        break;
    case Device.Windows:
        Label1.FontSize = Device.GetNamedSize(NamedSize.Large, Label1);
        break;
}
```

**RuntimePlatform** is of type **string** and can be easily compared against specific constants—**iOS**, **Android**, **WinPhone**, and **Windows**—that represent the supported platforms.

The **GetNamedSize** method automatically resolves the **Default**, **Micro**, **Small**, **Medium**, and **Large** platform font size and returns the corresponding **double**, which avoids the need to supply numeric values that would be different for each platform.

The **Device.Idiom** property allows you to determine if the current device the app is running on is a phone, tablet, or desktop PC (UWP only), and returns one of the values from the **TargetIdiom** enumeration:

```
switch(Device.Idiom)
{
    case TargetIdiom.Desktop:
        // UWP desktop
        break;
    case TargetIdiom.Phone:
        // Phones
        break;
    case TargetIdiom.Tablet:
        // Tablets
        break;
    case TargetIdiom.Unsupported:
        // Unsupported devices
        break;
}
```

You can also decide how to adjust UI elements based on the platform and idiom in XAML. Code Listing 32 demonstrates how to adjust the **Padding** property of a page, based on the platform.

*Code Listing 32*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:views="clr-namespace:App1.Views"
             x:Class="App1.Views.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness"
                iOS="0, 20, 0, 0"
                Android="0, 10, 0, 0"
                WinPhone="0, 10, 0, 0" />
    </ContentPage.Padding>
</ContentPage>
```

With the **OnPlatform** tag, you can specify a different property value based on the **iOS**, **Android**, and **WinPhone** platforms. The property value depends on the **x:TypeArguments** attribute, which represents the .NET type for the property, **Thickness** in this particular case. Similarly, you can also work with **OnIdiom** and the **TargetIdiom** enumeration in XAML.

***Tip: In iOS, it is best practice to set a page padding of 20 from the top, like in the previous snippet. If you don't do this, your page will overlap the system bar. Also, for iPhone X and higher models, you also need to take care of the safe area. For more info, take a look at my article on how to handle the safe area in Xamarin.Forms.***

### Device-based localization

The **Device** class is useful not only for fine-tuning the user interface according to the device, but also for other features, such as localization. For example, the **Device** class exposes the **FlowDirection** property that makes it easier to implement right-to-left localization, and whose value can be bound to the **FlowDirection** property of each view, like in the following example:

```
<ContentPage FlowDirection="{x:Static Device.FlowDirection}">
```

The **ContentPage**'s content will be displayed according to the localization information retrieved from the device. In addition to XAML, you can also work with this property in C# code.

## Working with the dependency service

Most of the time, mobile apps need to offer interaction with the device hardware, sensors, system apps, and file system. Accessing these features from shared code is not possible because their APIs have unique implementations on each platform.

Xamarin.Forms provides a simple solution to this problem that relies on the service locator pattern: in the shared project, you write an interface that defines the required functionalities. Then, inside each platform project, you write classes that implement the interface through native APIs. Finally, you use the **DependencyService** class and its **Get** method to retrieve the proper implementation based on the platform your app is running on.

For example, suppose your app needs to work with SQLite local databases. Assuming you have installed the sqlite-net-standard NuGet package in your solution, in the .NET Standard project, you can write the following sample interface called **IDatabaseConnection**, which defines the signature of a method that must return the database path:

```
public interface IDatabaseConnection
{
    SQLite.SQLiteConnection DbConnection();
}
```

***Tip: A complete walkthrough of using local SQLite databases in Xamarin.Forms is available on MSDN Magazine from the author of this ebook.***

At this point, you need to provide an implementation of this interface in each platform project, because file names, path names, and, more generally, the file system, are platform-specific. Add a new class file called **DatabaseConnection.cs** to the iOS, Android, and Windows projects.

Code Listing 33 provides the iOS implementation, Code Listing 34 provides the Android implementation, and Code Listing 35 provides the Windows implementation.

*Code Listing 33*

```
using System;
using SQLite;
using System.IO;
using App1.iOS;

[assembly: Xamarin.Forms.Dependency(typeof(DatabaseConnection))]
namespace App1.iOS
{
    public class DatabaseConnection : IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {
            string dbName = "MyDatabase.db3";
            string personalFolder =
              System.Environment.
              GetFolderPath(Environment.SpecialFolder.Personal);
            string libraryFolder =
              Path.Combine(personalFolder, "..", "Library");
            string path = Path.Combine(libraryFolder, dbName);
            return new SQLiteConnection(path);
        }
    }
}
```

*Code Listing 34*

```
using Xamarin.Forms;
using App1.Droid;
using SQLite;
using System.IO;

[assembly: Dependency(typeof(DatabaseConnection))]
namespace App1.Droid
{
    public class DatabaseConnection: IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {
            string dbName = "MyDatabase.db3";
            string path = Path.Combine(System.Environment.
              GetFolderPath(System.Environment.
              SpecialFolder.Personal), dbName);
            return new SQLiteConnection(path);
        }
    }
}
```

*Code Listing 35*

```
using SQLite;
using Xamarin.Forms;
using System.IO;
using Windows.Storage;
using App1.UWP;

[assembly: Dependency(typeof(DatabaseConnection))]
namespace App1.UWP
{
    public class DatabaseConnection : IDatabaseConnection
    {
        public SQLiteConnection DbConnection()
        {
            {
                string dbName = "MyDatabase.db3";
                string path = Path.Combine(ApplicationData.
                  Current.LocalFolder.Path, dbName);
                return new SQLiteConnection(path);
            }
        }
    }
}
```

Each implementation decorates the namespace with the **`Dependency`** attribute, assigned at the assembly level, which uniquely identifies the implementation of the **`IDatabaseConnection`** interface at runtime. In the **`DbConnection`** method body, you can see how each platform leverages its own APIs to work with filenames. In the .NET Standard project, you can simply resolve the proper implementation of the **`IDatabaseConnection`** interface as follows:

```
// Get the connection to the database.
SQLiteConnection
database = DependencyService.Get<IDatabaseConnection>().DbConnection();
```

The **DependencyService.Get** generic method receives the interface as the type parameter and resolves the implementation of that interface according to the current platform. With this approach, you do not need to worry about determining the current platform and invoking the corresponding native implementations, since the dependency service does the job for you. This approach applies to all native APIs you need to invoke and provides the most powerful option to access platform-specific features in Xamarin.Forms.

## Wrapping native APIs: Xamarin.Essentials

***Tip: All the code examples described in this chapter require a*** *`using Xamarin.Essentials`* ***directive.***

When accessing native APIs, most of the time your actual need is to access features that exist cross-platform, but with APIs that are totally different from one another. For example, iOS, Android, and Windows devices all have a camera, they all have a GPS sensor that returns the current location, and so on.

For scenarios in which you need to work with capabilities that exist cross-platform, you can leverage a library called Xamarin.Essentials, a free and open-source library that is automatically added to Xamarin.Forms solutions at creation. At the time of writing, the latest stable version available is 1.6.1. This library covers almost 40 features that exist cross-platform, such as sensors, location, network connection, communication, and much more. I will provide some examples next to help you understand how and where it can be so useful.

### Checking the network connection status

***Tip: On Android, you need to enable the*** *`ACCESS_NETWORK_STATE`* ***permission in the project manifest.***

One of the most common requirements in mobile apps is checking for the availability of a network connection. With Xamarin.Essentials, this is very easy:

```
if(Connectivity.NetworkAccess == NetworkAccess.Internet)
{
    // Internet is available
}
```

The **Connectivity** class provides everything you need to detect network connection and state. The **NetworkAccess** property returns the type of connection, with values from the **NetworkAccess** enumeration: **Internet**, **ConstrainedInternet**, **Local**, **None**, and **Unknown**. The **ConnectionProfiles** property from the **Connectivity** class allows us to understand the type of connection in more detail, as demonstrated in the following example:

```
var profiles = Connectivity.ConnectionProfiles;
if(profiles.Contains(ConnectionProfile.WiFi))
{
    // WiFi connection
}
```

This is very useful because it allows us to understand the following type of connections: **WiFi**, **Ethernet**, **Cellular**, **Bluetooth**, and **Unknown**. The **Connectivity** class also exposes the **ConnectivityChanged** event, which is raised when the status of the connection changes. You can declare a handler in the usual way:

```
Connectivity.ConnectivityChanged += Connectivity_ConnectivityChanged;
```

Then you can leverage the **ConnectivityChangedEventArgs** object to understand what happened and react accordingly:

```
private async void Connectivity_ConnectivityChanged(object sender,
                   ConnectivityChangedEventArgs e)
{
    if(e.NetworkAccess != NetworkAccess.Internet)
    {
        await DisplayAlert("Warning", "Limited internet connection", "OK");
        // Do additional work to limit network access...
    }
}
```

As you can see, this feature simplifies one of the most important tasks an app must perform, and that would otherwise require writing specific code for each platform.

## Opening URIs

It is common to include hyperlinks within the user interface of an app. Xamarin.Essentials provides the **Browser** class, which makes it simple to open URIs inside the system browser. You use it as follows:

```
await Browser.OpenAsync("https://www.microsoft.com");
```

Xamarin.Essentials also offers the **Launcher** class, which allows for opening any kind of URIs with the system default app. For example, the following code opens the default email client:

```
string uri = "mailto://somebody@something.com";
var canOpen = await Launcher.CanOpenAsync(uri);
if(canOpen)
    await Launcher.OpenAsync("mailto://somebody@something.com");
```

It is good practice to check if the system supports opening the specified URI via the **CanOpenAsync** method; if this returns **true**, you can then invoke **OpenAsync** to open the specified URI inside the default system app.

## Sending SMS messages

Sending SMS messages is very straightforward with Xamarin.Essentials. Look at the following code:

```
public async Task SendSms(string messageText, string[] recipients)
{
    var message = new SmsMessage(messageText, recipients);
    await Sms.ComposeAsync(message);
}
```

The **`SmsMessage`** class needs the message text and a list of recipients in the form of an array of string objects. Then the **`ComposeAsync`** from the **`Sms`** class will open the system UI for sending messages. Without Xamarin.Essentials, this would require working with the dependency service, and with three different platform-specific implementations. With Xamarin.Essentials, you accomplish the same result with two lines of code.

## Handling exceptions

If a feature is not available on the current device and system, Xamarin.Essentials's types raise a **`FeatureNotSupportedException`**. It is recommended to always handle this exception in a **`try..catch`** block to avoid unexpected app crashes.

## More Xamarin.Essentials

As you can imagine, it is not possible to provide examples for all the features wrapped by the Xamarin.Essentials library inside a book of the *Succinctly* series. The complete list of features with examples is available in the official documentation page, which is regularly updated when new releases are out.

# Working with native views

In previous sections, you looked at how to interact with native Android, iOS, and Windows features by accessing their APIs directly in C# code or through plugins. In this section, you will see how to use native views in Xamarin.Forms, which is extremely useful when you need to extend views provided by Xamarin.Forms, or when you wish to use native views that Xamarin.Forms does not wrap into shared objects out of the box.

## Embedding native views in XAML

Xamarin.Forms allows you to add native views directly into the XAML markup. This feature is a recent addition, and it makes it really easy to use native visual elements. To understand how native views in XAML work, consider Code Listing 36.

*Code Listing 36*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:ios="clr-namespace:UIKit;
                        assembly=Xamarin.iOS;targetPlatform=iOS"
             xmlns:androidWidget="clr-namespace:Android.Widget;
                        assembly=Mono.Android;targetPlatform=Android"
             xmlns:formsandroid="clr-namespace:Xamarin.Forms;
                                 assembly=Xamarin.Forms.Platform.Android;
                                 targetPlatform=Android"
             xmlns:win="clr-namespace:Windows.UI.Xaml.Controls;
                        assembly=Windows, Version=255.255.255.255,
                        Culture=neutral, PublicKeyToken=null,
                        ContentType=WindowsRuntime;targetPlatform=Windows"
             x:Class="App1.MainPage" Title="Native views">
    <ContentPage.Content>
        <StackLayout>
            <ios:UILabel Text="Native Text" View.HorizontalOptions="Start"/
>
            <androidWidget:TextView Text="Native Text"
                   x:Arguments="{x:Static formsandroid:Forms.Context}" />
            <win:TextBlock Text="Native Text"/>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

In the XAML of the root page, you first need to add XML namespaces that point to the namespaces of native platforms. The **`formsandroid`** namespace is required by Android widgets to get the current UI context. Remember that you can choose a different name for the namespace identifier. Using native views is then really simple, since you just need to declare the specific view for each platform you want to target.

In Code Listing 36, the XAML markup includes a **`UILabel`** native label on iOS, a **`TextView`** native label on Android, and a **`TextBlock`** native view on Windows. With Android views, you must supply the current Xamarin.Forms UI context, which is done with a special syntax that binds the static (**`x:Static`**) **`Forms.Context`** property to the view. You can interact with views in C# code as you would normally do, such as with event handlers, but the good news is that you can also assign native properties to each view directly in your XAML.

## Working with custom renderers

Renderers are classes that Xamarin.Forms uses to access and render native views, and that bind the Xamarin.Forms views and layouts discussed in Chapters 4 and 5 to their native counterparts.

For example, the **Label** view discussed in Chapter 4 maps to a **LabelRenderer** class that Xamarin.Forms uses to render the native **UILabel**, **TextView**, and **TextBlock** views on iOS, Android, and Windows, respectively. Xamarin.Forms views completely depend on renderers to expose their look and behavior. The good news is that you can override the default renderers with the *custom renderers*, which you can use to extend or override features in the Xamarin.Forms views. A custom renderer is a class that inherits from the renderer that maps the native view and is the place where you can change the layout, override members, and change the view's behavior. An example will help you better understand custom renderers.

Suppose you want an **Entry** view to autoselect its content when the user taps the text box. Xamarin.Forms has no support for this scenario, so you can create a custom renderer that works at the platform level. In the .NET Standard project, add a new class called **AutoSelectEntry** that looks like the following:

```
using Xamarin.Forms;
namespace App1
{
    public class AutoSelectEntry: Entry
    {
    }
}
```

The reason for creating a class that inherits from **Entry** is that, otherwise, the custom renderer you will create shortly would be applied to all the **Entry** views in your user interface. By creating a derived view, you can decide to apply the custom renderer only to this one. If you instead want to apply the custom renderer to all the views in the user interface of that type, you can skip this step.

The next step is creating a class that inherits from the built-in renderer (the **EntryRenderer** in this case) and provides an implementation inside each platform project.

***Note: In the next code examples, you will find many native objects and members. I will only highlight those that are strictly necessary to your understanding. The descriptions for all the others can be found in the Xamarin.iOS, Xamarin.Android, and Universal Windows Platform documentation.***

Code Listing 37 shows how to implement a custom renderer in iOS, Code Listing 38 shows the Android version, and Code Listing 39 shows the Windows version.

*Code Listing 37*

```
[assembly: ExportRenderer(typeof(AutoSelectEntry),
           typeof(AutoSelectEntryRenderer))]
namespace App1.iOS
{
    public class AutoSelectEntryRenderer: EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<En
try> e)
```

```
        {
            base.OnElementChanged(e);
            var nativeTextField = Control;
            nativeTextField.EditingDidBegin += (object sender, EventArgs eI
os) =>
            {
                nativeTextField.PerformSelector(new ObjCRuntime
                               .Selector("selectAll"),
                null, 0.0f);
            };
        }
    }
}
```

*Code Listing 38*

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;
using NativeAccess;
using NativeAccess.Droid;

[assembly: ExportRenderer(typeof(AutoSelectEntry),
 typeof(AutoSelectEntryRenderer))]
namespace App1.Droid
{
    public class AutoSelectEntryRenderer: EntryRenderer
    {

        public AutoSelectEntryRenderer(Context context): base(context)
        {

        }

        protected override void OnElementChanged(ElementChangedEventArgs<En
try> e)
        {
            base.OnElementChanged(e);
            if (e.OldElement == null)
            {
                var nativeEditText = (global::Android.Widget.EditText)Contr
ol;
                nativeEditText.SetSelectAllOnFocus(true);
            }
        }
    }
}
```

*Code Listing 39*

```
using App1;
using App1.UWP;
using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ExportRenderer(typeof(AutoSelectEntry),
           typeof(AutoSelectEntryRenderer))]
namespace App1.UWP
{
    public class AutoSelectEntryRenderer: EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<En
try> e)
        {
            base.OnElementChanged(e);
            if (e.OldElement == null)
            {
                var nativeEditText = Control;
                nativeEditText.SelectAll();
            }
        }
    }
}
```

In each platform implementation, you override the **`OnElementChanged`** method to get the instance of the native view via the **`Control`** property, and then you invoke the code necessary to select all the text box content using native APIs. The **`ExportRenderer`** attribute at the assembly level tells Xamarin.Forms to render views of the specified type (**`AutoSelectEntry`** in this case) with an object of type **`AutoSelectEntryRenderer`**, instead of the built-in **`EntryRenderer`**. Once you have the custom renderer ready, you can use the custom view in XAML as you would normally do, as demonstrated in Code Listing 40.

*Code Listing 40*

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:App1"
             Title="Main page"
             x:Class="App1.MainPage">

    <StackLayout Orientation="Vertical" Padding="20">
        <Label Text="Enter some text:"/>

        <local:AutoSelectEntry x:Name="MyEntry" Text="Enter text..."
        HorizontalOptions="FillAndExpand"/>
    </StackLayout>
</ContentPage>
```

***Tip: The `local` XML namespace is defined by default, so adding your view is even simpler. Additionally, IntelliSense will show your custom view in the list of available objects from that namespace.***

If you now run this code, you will see that the text in the **`AutoSelectEntry`** view will be automatically selected when the text box is tapped. Custom renderers are very powerful because they allow you to completely override the look and behavior of any views. However, sometimes you just need some minor customizations that can instead be provided through effects.

## Hints for effects

Effects can be thought of as simplified custom renderers, limited to changing some layout properties without changing the behavior of a view. An effect is made of two classes: a class that inherits from **`PlatformEffect`** and must be implemented in all the platform projects; and a class that inherits from **`RoutingEffect`** and resides in the .NET Standard (or shared) project, whose responsibility is resolving the platform-specific implementation of the custom effect. You handle the **`OnAttached`** and **`OnDetached`** events to provide the logic for your effect. Because their structure is similar to custom renderers' structures, I will not cover effects in more detail here, but it is important you know they exist. You can check out the official documentation, which explains how to consume built-in effects and create custom ones.

## Introducing platform-specifics

Xamarin.Forms provides the platform-specifics, which allow for consuming features that are available only on specific platforms. Platform-specifics represent a limited number of features, but they allow you to work without implementing custom renderers or effects.

***Note: Platform-specifics do not represent features that are available cross-platform. They instead provide quick access to features that are available only on specific platforms. As an additional clarification, a platform-specific might be available on iOS, while the same platform-specific might not exist for Android and UWP.***

For instance, suppose you are working on an iOS app and you want the separator of a **`ListView`** to be full width (which is not the default). Without platform-specifics, you would need to implement a custom renderer to accomplish this. With platform-specifics, you just need the code shown in Code Listing 41.

*Code Listing 41*

```
<?xml version="1.0" encoding="utf-8" ?>
```

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.
Forms.Core"
             x:Class="NativeAccess.PlatformSpecificsPage">
    <ContentPage.Content>
        <StackLayout>
            <ListView ios:ListView.SeparatorStyle="FullWidth"
             x:Name="ListView1">
                <!-- Bind your data and add a data template here... -->
            </ListView>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

In the case of iOS, you need to import the **`Xamarin.Forms.PlatformConfiguration.iOSSpecific`** namespace. Then you can use attached properties provided by this namespace on the view of your interest. In the example shown in Code Listing 37, the attached property **`ListView.SeparatorStyle`** allows you to customize the separator width.

Platform-specifics can also be used in C# code. In this case, you need two **`using`** directives to import the **`Xamarin.Forms.PlatformConfiguration`** and **`Xamarin.Forms.PlatformConfiguration.iOSSpecific`**. Then you can invoke the **`On`** method on the view of your interest, passing the target platform, and supplying the platform-specific implementation you need. The following code provides an example that represents the same scenario seen in Code Listing 37, but in C# code:

```
this.ListView1.On<iOS>().SetSeparatorStyle(SeparatorStyle.FullWidth);
```

Platform-specifics work the same way on Android and UWP. In the case of Android, the namespace you import is **`Xamarin.Forms.PlatformConfiguration.AndroidSpecific`** (for both XAML and C#), whereas for UWP the namespace is **`Xamarin.Forms.PlatformConfiguration.WindowsSpecific`**. The list of built-in platform-specifics in Xamarin.Forms is available in the documentation.

***Tip: A platform-specific for one platform will simply be ignored on other platforms.***

## Chapter summary

Mobile apps often need to work with features that you can only access through native APIs. Xamarin.Forms provides access to the entire set of native APIs on iOS, Android, and Windows via a number of possible options. With the **`Device`** class, you can get information on the current system from your shared code. With the **`DependencyService`** class and its **`Get`** method, you can resolve cross-platform abstractions of platform-specific code in your .NET Standard library.

With Xamarin.Essentials, you have ready-to-use cross-platform abstractions for the most common scenarios, such as (but not limited to) accessing sensors, the network information, settings, or battery status. In terms of native visual elements, you can embed iOS, Android, and Windows native views directly in your XAML. You can also write custom renderers or effects to change the look and feel of your views, and you can use platform-specifics to quickly implement a few features that are available only on specific platforms. Actually, each platform also manages the app lifecycle with its own APIs. Fortunately, Xamarin.Forms has a cross-platform abstraction that makes it simpler, as explained in the next chapter.

# Chapter 10 Managing the App Lifecycle

The application lifecycle involves events such as startup, suspend, and resume. Every platform manages the application lifecycle differently, so implementing platform-specific code in iOS, Android, and Windows projects would require some effort. Luckily, Xamarin.Forms allows you to manage the app lifecycle in a unified way and takes care of performing the platform-specific work on your behalf. This chapter provides a quick explanation of the app lifecycle and of how you can easily manage your app's behavior.

## Introducing the App class

The **App** class is a singleton class that inherits from **Application** and is defined inside the **App.xaml.cs** file. It can be thought of as an object that represents your application running and includes the necessary infrastructure to handle resources, navigation, and the application lifecycle. If you need to store some data in variables that should be available to all pages in the application, you can expose static fields and properties in the **App** class.

At a higher level, the **App** class exposes some fundamental members that you might need across the whole app lifecycle: the **MainPage** property you assign with the root page of your application, and the **OnStart**, **OnSleep**, and **OnResume** methods you use to manage the application lifecycle that are described in the next section.

## Managing the app lifecycle

The application lifecycle can be summarized in four events: startup, suspension, resume, and shutdown. The Android, iOS, and Windows platforms manage these events differently, but Xamarin.Forms provides a unified system that allows for managing an app's startup, suspension, and resume from a single, shared C# codebase. These events are represented by the **OnStart**, **OnSleep**, and **OnResume** methods that you can see in the **App.xaml.cs** file, whose body is empty.

Currently, no specific method handles the app shutdown, because in most cases handling suspension is sufficient. For instance, you might load some app settings within **OnStart** at startup, save settings when the app is suspended within **OnSleep**, and reload settings when the app comes back to the foreground within **OnResume**. It is common to store the date and time of the last activity, for example. The Xamarin.Essentials library provides the **Preferences** class, which makes this very simple. If you look at Code Listing 42, you can see how it works.

*Code Listing 42*

```
using App1.Helpers;
using System;
using Xamarin.Forms;

namespace App1
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();

            MainPage = new App1.MainPage();
        }

        private DateTime _lastActivityTime;
        protected override void OnStart()
        {
            _lastActivityTime =
             Preferences.Get("LastActivityTime", DateTime.MinValue);
        }

        protected override void OnSleep()
        {
            Preferences.Set("LastActivityTime",
            DateTime.Now);
        }

        protected override void OnResume()
        {
            _lastActivityTime = Preferences.Get("LastActivityTime",
             DateTime.MinValue);
        }
    }
}
```

The **`Set`** method stores preferences in the form of a key/value pair. In this case, the key is an identifier used to store and retrieve the date and time of the last activity. The **`Get`** method returns the value for the specified key and allows for specifying a default value if the key is not found. In this example, the date and time of the last activity is stored at app suspension and retrieved at app startup or resume.

## Sending and receiving messages

Xamarin.Forms includes an interesting static class called **`MessagingCenter`**. This class can send broadcast messages that subscribers can receive and take actions, based on a publisher/subscriber model. In its most basic form, you use the **`MessagingCenter`** to send a message as follows:

```
MessagingCenter.Send<MainPage>(this, "MESSAGE");
```

The **`Send`** method's type parameters specify the types subscribers should expect, and its arguments are the sender (**`MainPage`** in this case, as an example) and the message in the form of a string. You can specify multiple type parameters, and therefore multiple arguments before the message.

***Tip: The compiler is able to infer type parameters for `Send`, so it is not mandatory to specify them explicitly.***

Subscribers can then listen for messages and take actions as follows:

```
MessagingCenter.Subscribe<MainPage>
    (this, "MESSAGE", (sender) =>
    {
        // Do something here.
    });
```

When **`MessagingCenter.Send`** is invoked somewhere, objects listening for a particular message will execute the action specified within **`Subscribe`** (this does not have to necessarily be a lambda expression; it can be an expanded delegate). When their job is finished, subscribers can invoke **`MessagingCenter.Unsubscribe`** to stop listening to a message, passing the sender as the type parameter, the current object, and the message, as follows:

```
MessagingCenter.Unsubscribe<MainPage>(this, "MESSAGE");
```

The **`MessagingCenter`** class can be very useful when you have logics that are decoupled from the user interface, and it can even be useful with MVVM implementations.

## Chapter summary

Managing the application lifecycle can be very important, especially when you need to get and store data at the application startup or suspension. Xamarin.Forms prevents the need to write platform-specific code and offers a cross-platform solution through the **`OnStart`**, **`OnSleep`**, and **`OnResume`** methods that allow handling the startup, suspension, and resume events, respectively, from a single C# codebase—regardless of the platform the app is running on. Not only is this a powerful feature, but it really simplifies your work as a developer.

Finally, you have seen in this chapter the **`MessagingCenter`** class, a static object that allows for sending and subscribing messages, which is useful with logics decoupled from the user interface.